Bixby Modeling

At first glance, Bixby can seem difficult to develop for compared to other voice assistants. It has its own development environment! It even has its own language!

More confusingly, programs (or "capsules") for Bixby aren't constructed anything like programs for other voice assistants you might be more familiar with. Instead, Bixby throws unfamiliar ideas at you: you model concepts and actions, train utterances (the user's natural language input) to goals, then let Bixby itself plan and execute a dynamically generated program. This doesn't sound like an easier way to write a program!

Well, it does depend on the program. But most of the time, capsule development with Bixby Developer Studio is not only more powerful, it's easier than the alternatives.

Modeling vs. Directive

Bixby isn't difficult once you understand its paradigm. To get a better handle on that, let's *compare modeling*, Bixby's approach, to *Alexa's directive approach*.

The Directive Approach

All computer programs are essentially a list of steps taken in order, with the ability to repeat sets of steps (looping) and to skip over or include steps based on testing inputs or past results (conditionals).

Alexa has a fairly straightforward model for programming "skills" that we'll call a *directive approach*. Your skill consists of related actions; a weather skill might have an action for returning the temperature at a specific time in a specific location, another action for returning the high temperature expected in a specific location on a specific date, and so on. (Alexa calls these actions "intents" if you're creating a Custom Skill—most analogous to a Bixby capsule—but calls them "directives" when you're using a "pre-built model," a fixed, unmodifiable set of actions for a specific domain like controlling a media player or performing simple query-and-response actions.)

This is a rough checklist for programming an Alexa skill:

1. Define optional arguments (Alexa calls them "slots") for the intent. Slots have:
   - Prompts for the user to give required values if they weren't in the initial utterance
   - Optional default values
   - Validation rules
   - Optional confirmations (that is, Alexa must confirm it understands the value correctly before continuing)
2. Define one or more utterances for the intent. These utterances are unique sentences that call that intent, and might fill in values for slots. For example, "what will the weather be like in Sacramento on Tuesday" matches a `GetWeatherReport` intent with two slots, `location` ("Sacramento") and `date` ("Tuesday").
   - If values for required slots aren't supplied, Alexa will use default values if they're provided.
   - If there's no default for a required slot, Alexa will prompt the user for a value.
3. The intent defines a function that takes the inputs, performs any actions (such as calling an external web service), and returns a value.

Now, this looks simple, but there's a lot of work involved. The intent described above, `GetWeatherReport`, has to handle all these utterances:

- "Get the weather in Sacramento on Tuesday"
- "Get the weather on Tuesday"
- "Get the weather in Sacramento"
- "Get the weather"

Only the first utterance specifies a date and location. The default date is (probably) today; the default location is (probably) the user's location. Your function will have to include code to handle these.

(N.B.: In practice, there's a set of provided weather intents for Alexa that would do the work for you in this example, but roll with it.)

The Modeling Approach

Let's take the same utterance, "what will the weather be like in Sacramento on Tuesday," as our example. We're going to define a `GetWeatherReport` action in Bixby with a location and a date concept tagged, and then we'll define a function that takes those as its inputs. Just like Alexa! Right?

Well...no.

The directive approach puts its focus on that function. It's a code-centric way for the voice assistant to get an answer. Bixby, though, has a model-centric way to get one. It focuses on the concepts and actions, providing a rich modeling language that cuts down—sometimes dramatically—on the amount of code you'll need to write to let Bixby do the same task as Alexa.

In the modeling approach, you start by defining the models for your weather report capsule. Let's think about concepts in a weather report:

- The date of the report
- The location of the report
- The high and low temperatures for that day and location
- The expected amount and kind of precipitation for that day and location

You define these using a combination of *primitive data types* (integers, booleans, text, and so on) and structure concepts that store primitives—and other structures—as properties, like a JSON object or C structure. Bixby's library includes clever structures for times and dates, locations, and even measurements including units; those cover everything we need except types of precipitation. Let's create a new concept in Bixby's modeling language using the enum (enumeration) primitive:

```
enum (PrecipitationType) {
  symbol (Rain)
  symbol (Snow)
  symbol (Hail)
  symbol (Sleet)
  symbol (Mix)
  symbol (None)
}
```

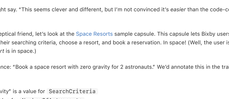
Building on that, here's a possible `WeatherReport` concept model. This uses concepts from Bixby's library to handle temperature, "named-point" (locations that Bixby can look up by name, like "Sacramento" or "JFK airport"), and a date/time stamp.

```
structure (WeatherReport) {
  description (The weather report)
  property (date) {
    type (time.timeDate)
  }
  property (location) {
    type (geo.NamedPoint)
  }
  property (highTemp) {
    type (measurement.Temperature)
  }
  property (lowTemp) {
    type (measurement.Temperature)
  }
  property (precipitationAmount) {
    type (measurement.Height)
  }
  property (precipitationType) {
    type (PrecipitationType)
  }
}
```

A more advanced model could store the weather forecast hour by hour, and include other data like wind speed, air quality, UV index, humidity, dew point, and more. We could model that with an `HourlyReport` structure, which could be stored in a property within `WeatherReport`. Properties have cardinality in Bixby: they can be `Required` or `Optional`, and can hold either `One` or `Many` values.

Now that you have a `WeatherReport` concept, you'd create an action model for `GetWeatherReport` that takes `date` and `location` as inputs and returns a `WeatherReport` structure. It declares two `input` blocks and one `output` block, and then declares an endpoint that lets Bixby know where the JavaScript code that executes the action is. (In Bixby Developer Studio, JavaScript files go in the `code/` directory.) This code receives JSON objects that contain the values of the input concepts, and returns a JSON object for the WeatherReport. Depending on how you define your models, your code might not have to do anything beyond calling your weather service's API if it's also JSON-based.

The last step with Bixby is training. This is where you enter utterances—such as our example "what will the weather be like in Sacramento on Tuesday?"—and, using the models you've created, teach Bixby how to interpret them. Have you wondered why Bixby has its own dedicated development environment, Bixby Developer Studio? This is one of the answers: you create a training file directly within Studio, with its own GUI-editing tools.



We've entered our utterance here and marked "Sacramento" and "Tuesday" as values for their respective concepts (`location` and `date`), and entered a goal of `GetWeatherReport`.

So, is a "goal" the way Bixby associates an utterance with an action? In a general sense, yes, but it's not that simple a relationship. This goes back to Bixby's dynamic program generation. By annotating this utterance and giving it a goal, you're telling Bixby the starting point (the input values) and the ending point (the goal) of the program it needs to write to execute that utterance.

In this example, the program it's going to write is straightforward. But what about the utterances that leave out information?

- "Get the weather on Tuesday"
- "Get the weather in Sacramento"

Again, this requires the voice assistant to make assumptions about reasonable defaults (the weather in the user's location on Tuesday, or the weather in Sacramento today). With a code-centric approach like Alexa's, you'd need to write code to handle those cases. With Bixby's model-centric approach, we can do this entirely in the action model by specifying a default initialization value. This can incorporate system variables like the user's location. This is an `input` block from our `GetWeatherReport` action that sets the value of `geo.NamedPoint` to the user's current location by default:

```
input (location) {
  type (geo.NamedPoint)
  default-init {
    if ($user.currentLocation.$exists) {
      intent {
        goal: geo.NamedPoint
        value-set {
          geo.CurrentLocation: $expr($user.currentLocation)
        }
      }
    }
  }
}
```

Notice the `intent` and `goal` keywords: Bixby sets `location` to a default, the same way it processes utterances. In this case, it takes the value of `$user.currentLocation` as input and sets a new `geo.NamedPoint` as the goal. You didn't need to write a line of JavaScript to get here!

The Strength of Modeling

"Okay," you might say. "This seems clever and different, but I'm not convinced it's *easier* than the code-centric approach."

All right, my skeptical friend, let's look at the Space Resorts sample capsule. This capsule lets Bixby users search for resorts, refine their searching criteria, choose a resort, and book a reservation. In space! (Well, the user is here on Earth. The resort is in space.)

Take this utterance: "Book a space resort with zero gravity for 2 astronauts." We'd annotate this in the training like this:

- "zero gravity" is a value for `SearchCriteria`
- "2" is a value for `NumberOfAstronauts`
- The goal is `CommitOrder`

If you look at the models for Space Resort, `CommitOrder` takes one input, `Order`, and outputs a `Receipt`. Huh. What's going on there? Well, `Order` is a structure, just like our example `WeatherReport` above, but it only has two properties! One of those, `Item`, is the actual reservation: it requires the number of astronauts, a specific resort to book, the room type (er, "habitat pod" type), and the booking dates.

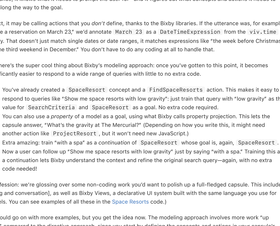
We only have one of those in that utterance. We'll have to write a lot of code to prompt for all that missing information, *right*? And code that creates that `Item` object and the `Order` object, and code that links that code together...

But in fact, we won't have to create as much code as you think. We will have to create actions like `CreateItem` and `CreateOrder`, and a bit of JavaScript for each. The JavaScript is pretty simple; for `CreateItem.js` it's just this:

```
var dates = require('dates')
module.exports.function = function (spaceResort, numberOfAstronauts, dateInterval, pod) {
  return {
    pod: pod,
    spaceResort: spaceResort,
    numberOfAstronauts: numberOfAstronauts,
    dateInterval: {
      start: dates.ZonedDateTime.fromDate(dateInterval.start).getDateTime().date,
      end: dates.ZonedDateTime.fromDate(dateInterval.end).getDateTime().date
    }
  }
}
```

The models do the heavy lifting, defining required inputs, setting default values, and even leveraging personalization and learning techniques. For instance, Bixby can remember that you've booked pods for three astronauts before, and pre-selects "3" if you don't specify a different number. (Naturally, you can always change it to a different number before booking.)

So how do you wire all these models up to get from the values in the utterance to that final CommitOrder action? You don't. *Bixby does.*



That's the execution graph Bixby creates for our example utterance, starting at the top with our natural language (NL) input and ending at the bottom with the goal. Because Bixby understands the models and the values in the utterance, it knows what it needs to prompt the user for—and it figures out what concepts and actions it needs to call along the way to the goal.

In fact, it may be calling actions that you *don't* define, thanks to the Bixby libraries. If the utterance was, for example, "make a reservation on March 23," we'd annotate `March 23` as a `DateTimeExpression` from the `viv.time` library. That doesn't just match single dates or date ranges, it matches expressions like "the week before Christmas" or "the third weekend in December." You don't have to do any coding at all to handle that.

And here's the super cool thing about Bixby's modeling approach: once you've gotten to this point, it becomes significantly easier to respond to a wide range of queries with little to no extra code.

- You've already created a `SpaceResort` concept and a `FindSpaceResorts` action. This makes it easy to respond to queries like "Show me space resorts with low gravity": just train that query with "low gravity" as the value for `SearchCriteria` and `SpaceResort` as a goal. No extra code required.
- You can also use a *property* of a model as a goal, using what Bixby calls property projection. This lets the capsule answer, "What's the gravity at The Mercurial?" (Depending on how you write this, it might need another action like `ProjectResort`, but it won't need new JavaScript.)
- Extra amazing: train "with a spa" as a constraint of `SpaceResort` whose goal is, again, `SpaceResort`. Now a user can follow up "Show me space resorts with low gravity" just by saying "with a spa." Training this as a continuation lets Bixby understand the context and refine the original search query—again, with no extra code needed!

(Confession: we're glossing over some non-coding work you'd want to polish up a full-fledged capsule. This includes dialog and conversation), as well as Bixby Views, a declarative UI system built with the same language you use for models. You can see examples of all these in the Space Resorts code.)

We could go on with more examples, but you get the idea now. The modeling approach involves more work "up front" compared to the directive approach, since you start by defining the concepts and actions in your capsule's domain rather than starting by planning out utterances. Your reward as a developer, though, is a much richer set of possible interactions for your users, made available with almost no extra work on your part.

And cookies. With all the work you're saving, you definitely have time to reward yourself with cookies.