Version: 1.0

SIML Quick Start

*Only a **few** simple SIML features have been touched briefly in this presentation. To get into more details please visit* Developer Network

Any sufficiently advanced technology is indistinguishable from magic.

- Arthur C. Clarke.

Getting Started

What's this?

This is a quick start tutorial on SIML bot creation

By the end of this tutorial you will be able to create your own SIML Bot with simple functionalities

So grab a cup of coffee...

But before you begin your development download Chatbot Studio

- **Syn Chatbot Studio** is the official IDE for SIML Bot development.

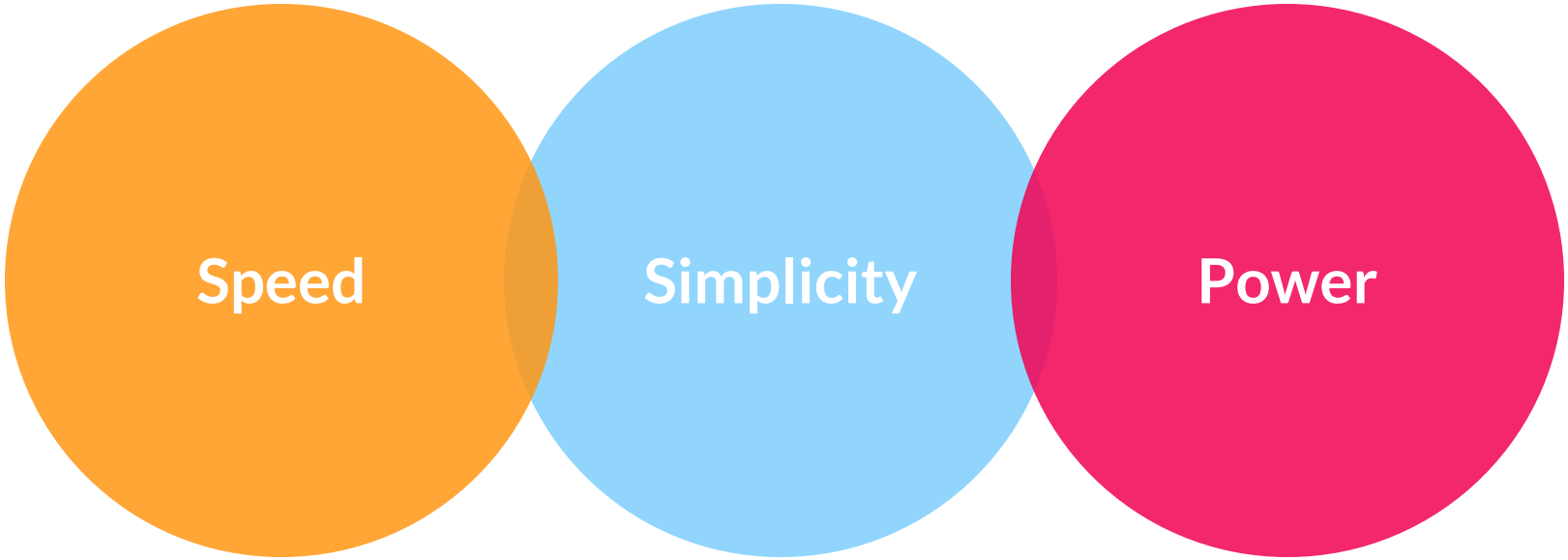
- To download Chatbot Studio visit SimlBot.com

System Requirements

All of the following system requirements should be met in their entirety

- **Windows 7 or Above**
- **.Net Framework 4.5**
- **1 GB RAM**
- **1 GB Free Disk Space**
- **Minimum 1024x768 Desktop Resolution**

Why SIML ? In three simple Words



500,000 Models

SIML Bots can load half a million Models in 4 Seconds

1 Unique Language

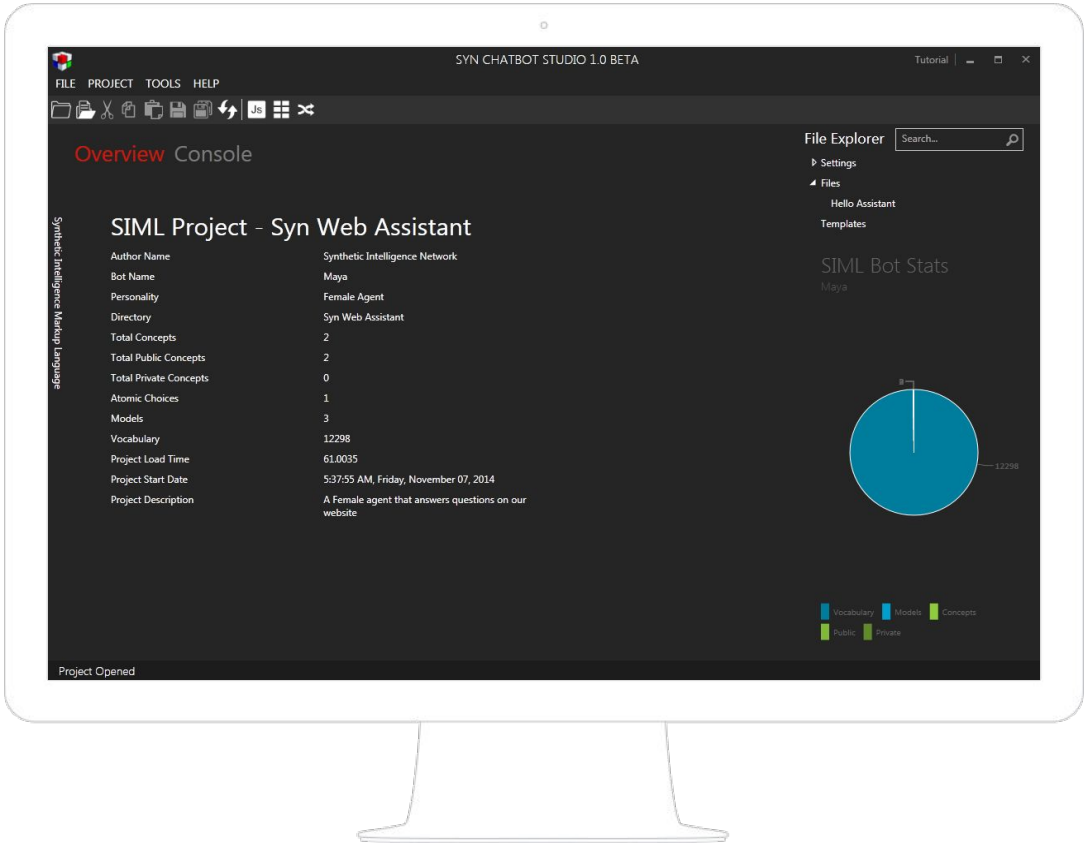
SIML is the only Chatbot Language that offers such a wide range of **unparalleled** features.

100%

Freedom

NEW PROJECT

Let’s create your first SIML Bot



Syn Chatbot Studio

- Click File->New->Project
- Fill in the details and select "**Syn Web Assistant**" as the base template and click "**Create Project**"
- Type in a filename and select a directory to save your new SIML Project to and click "**Save**"

Done!

You've now created your first SIML Project

Whoa! well that was easy.

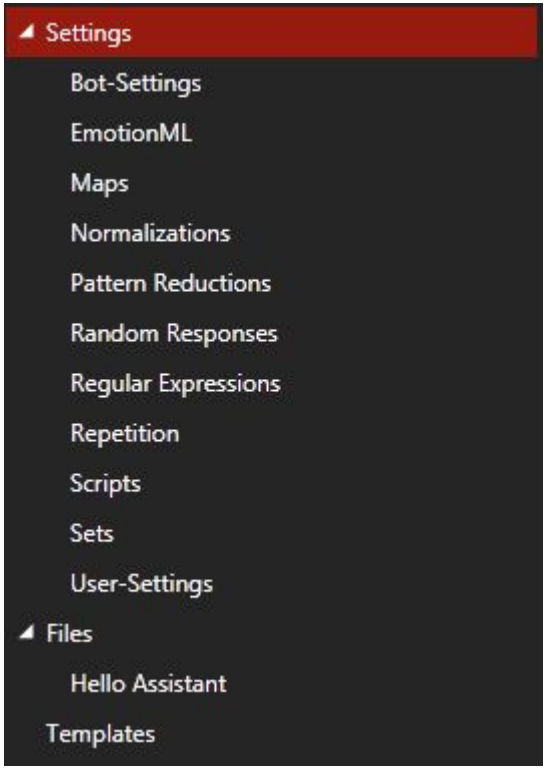
SIML Files Explorer

Settings

- Bot-Settings
- EmotionML
- Maps
- Normalizations
- Pattern Reductions
- Random Responses
- Regular Expressions
- Repetition
- Scripts
- Sets
- User-Settings

Files

- Hello Assistant



We will go through all of the above files in this presentation but the order of evaluation may not be the same.

Hello Assistant

Now that you have created your Project lets Chat with your new Bot.

Click on the “Console” tab and type “Hello Assistant”

Output

“Hello User!”

Explanation

When you type “**Hello Assistant**” a simple pattern is matched and a response is generated

SIML Code

```
<Siml>
  <Concept Type="public" Name="Hello Assistant">
    <Model>
      <Pattern>HELLO ASSISTANT</Pattern>
      <Response>Hello User!</Response>
    </Model>
  </Concept>
</Siml>
```

❊ SIML patterns are case-insensitive therefore if you would’ve typed “hello ASSISTANT” the response would still be the same.

Writing an SIML Model

➔ Click on “Hello Assistant” file under the “Files” category
➔ Type in the following SIML Code within the Concept “Hello Assistant”

SIML Code

```
<Model>
  <Pattern>
    <Item>HOW ARE YOU</Item>
    <Item>HOW IS IT GOING</Item>
  </Pattern>
  <Response>I am doing great!</Response>
</Model>
```

➔ Click on the Refresh button on the Toolbar or press F5

❊ Use Alt+M to insert a Model template
❊ A refresh is required to reload your changes and calibrate the Graph that SIML uses.

Your SIML Code

Your *new* SIML Code should look like the following within the Editor

SIML Code

```
<Siml>
  <Concept Type="public" Name="Hello Assistant">
    <Model>
      <Pattern>HELLO ASSISTANT</Pattern>
      <Response>Hello User!</Response>
    </Model>

    <Model>
      <Pattern>
        <Item>HOW ARE YOU</Item>
        <Item>HOW IS IT GOING</Item>
      </Pattern>
      <Response>I am doing great!</Response>
    </Model>

  </Concept>
</Siml>
```

Testing your Bot

Click on the “Console” tab and type “how are you”

Output

“I am doing great!”

Explanation

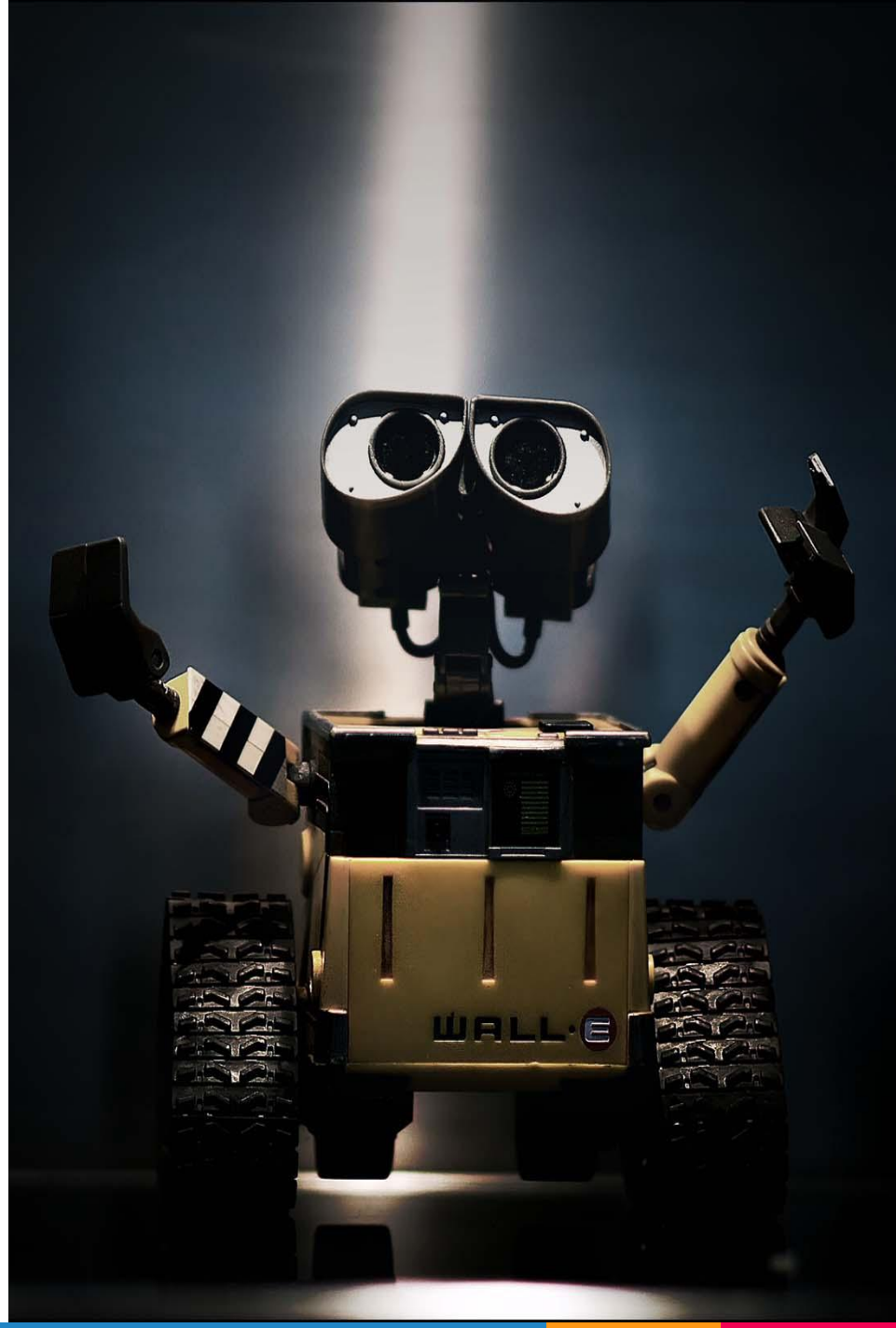
The SIML code you just wrote now has a model with patterns “how are you” and “how is it going” that yields a response “I am doing great!”. So when the user types “how are you” this model is activated and its response is evaluated. Note the <Item> element allows multiple patterns to be declared within the same element.

We will now learn about Concepts and Models

SIML Concepts

A Concept is a collection of Models.

A Concept is “your” abstract idea of how the user perceives a topic while interacting with your Bot.



Declaring a Concept

A SIML Concept has the following attributes

Type

Public or Private.

Models within a public Concept are visible globally and will be activated whenever their patterns match.

Models within a private Concept are not visible unless a Response within a public Model activates it.

Name

Every Concept in SIML should have a name.

A name is what identifies a Concept and tells the Bot what subject is under consideration.

A Concept may at times be synonymous to a Topic.

Repeat

True or False

Models within a Concept may or may not repeat themselves. i.e. they may be designed to be evaluated only once.

In that case a Concept is declared with an attribute Repeat=”False”

SIML Model

Basic unit of knowledge.

SIML Models

A basic unit of knowledge in SIML is stored within a Model

Pattern

A Pattern element holds textual patterns (simple or complex) that are checked when an input is received

Once a pattern is matched the Response element within the Model is evaluated.

Previous

Optionally the previous utterance of the Bot may be checked after a pattern is matched using the Previous tag.

If the previous utterance of the Bot is equal to the one specified within the Previous element then the Response element within the Model is evaluated

Response

The Response element within the Model is the part that is evaluated if it's ancestor pattern matched the user input.

A Response element may contain other SIML elements that transform the response that is sent to the user.

❊ Models can only occur within Concepts and other Models

Typical example of an SIML Model

SIML Code

```
<Model>
  <Pattern>YES</Pattern>
  <Previous>Do you like Chatbots</Previous>
  <Response>That is just awesome!</Response>
</Model>
```

- ➔ “YES” is the pattern
- ➔ “Do you like Chatbots” is last utterance of the Bot
- ➔ “That is just awesome!” is the response that will be generated if the user types “yes” after the bot says “Do you like Chatbots”

❊ Use Alt+M to insert a Model template when using Syn Chatbot Studio

❊ All SIML tags are **case-sensitive**

SIML Pattern

A sequence of SIML tokens that activates a Model

SIML Pattern

A Pattern is a sequence of SIML tokens or elements that get a Model activated

Atomic

A Pattern that comprises of only words. Words maybe in any language.

Non-Atomic|Complex

A Pattern that consists of words and/or pattern symbols. There are a number of SIML Pattern elements/wildcards that can be used to declare a non-atomic pattern

Conditional

A Pattern that consists an <If> element that specifies a condition to be evaluated

Scripted

A Pattern that consists a condition specified in a scripting language that, when evaluated, returns True or False

❊ Conditional and Scripted patterns shouldn't be used inside Public Concepts

Pattern Elements in SIML

1. **^** (up arrow) - Is prefixed to a word and makes SIML Bot search for the preceding word before any other pattern element. Example "**^Run Console**" will match "***Run Console***"

2. **%** (percent) - Matches zero or more words. Example "**Do you like %**" will match "***Do you like cupcakes***" or "***Do you like bacon***"

3. **_** (Underscore) - Matches one or more words. Example "**What is _**" will match "***What is a conscientious man***" or "***what is your name***"

4. **Atomic -** These are word based patterns and contain no symbols. Example "**How are you**" will match exactly "***how are you***" or "***HOW are YoU***" as user input is case-insensitive to an SIML Bot.

Pattern Elements in SIML

5. **@Regex -** Denotes an pre-defined regular expression. Example "**Switch @Switch**" where Switch can be any regular expression say with the pattern "**\b(on|off)\b**" and will match the sentence "**Switch on**" or "**Switch Off**"

6. **$** (dollar) - Matches one or more words but has a lower priority than %. Example "**What do you $**"

7. **{Word1 Word2 Word3}** - Is a keywords based pattern and matches the word in the given order even if intermediary words exists within the specified sequence of words. Example "**{what your name}**"will match "**what is your name**"

8. **[SET]** - A Set is a collection of words stored in the Settings File. Example "**I like the color [COLOR]**" will match sentences like "**I like the color red**" , "**I like the color blue**" or "**I like the color green**"

Pattern Elements in SIML

9. (Word1|Word2|Word3) - A Dynamic set that stores related words. Example "**I like the color (Red|Green|Blue**)" will match "***I like the color red***" or "***I like the color green***" or even "***I like the color blue***

10. **<If>** (Conditional) **-** a conditional pattern in SIML is activated if the condition within the **If** element is satisfied or returns true. Example **<If User="Age" Value="26"/>** will match if the age of the user is 26 and will activate the underlying Model.

11. **<Js>** (Scripted) - A Scripted pattern contains a code in a scripting language and if True is returned upon evaluation the Model is activated. Example **<Js>isValid();</Js>**. SIML Bot offers JavaScript as the default Scripting Language and has an in-built JavaScript interpreter.

12. * (Star) - Matches one or more words and has the lowest priority in the evaluation tree. Example pattern would "How are you *" will match "**How are you today**" but wouldn't match "**how are you**"

SIML Response

A Response is the final output a bot returns after evaluating an input

Responses may or may not contain a textual value and encapsulate one or more SIML elements

Creating a Concept and a Model with patterns

➔ Right click on “Files” under *Files Explorer* and select “Add New File”
➔ Type “My First Concept” and press OK

SIML Code

```
<Siml>
  <Concept Name="Hello Assistant" Type="public" Repeat="True" ></Concept>
</Siml>
```

➔ Click just before *</Concept>* and press Alt+M to insert a new Model template
➔ Type the following SIML Code

SIML Code

```
<Model>
  <Pattern>WHAT IS A *</Pattern>
  <Response>What do you think a <Match /> is?</Response>
</Model>
```

❊ Press Alt+I to indent the SIML document

Test your SIML Code

➔ Click on the Console tab and type “what is a Car”
➔ The Bot replies “What do you think a Car is?

Wildcard

Whenever you use a wildcard one or more word are matched and a value is stored in an indexed stack.

<Match/>

To access the captured word or sets of words you may use the **Match** tag. The first *capture* is stored at Index 1 and then Index 2, Index 3 and so on. The Index can be accessed using the “**Index**” attribute.

SIML Code (Example)

```
<Model>
  <Pattern>Does * like *</Pattern>
  <Response>Yes <Match /> likes <Match Index="2"/></Response>
</Model>
```

For the above code if the user writes “**Does Sarah like nascar**” the Bot will reply “***Yes Sarah likes nascar***”

❊ Press Alt+I to indent the SIML document

Keywords as patterns

➔ Type the following SIML Code within your newly created concept

SIML Code

```
<Model>
  <Pattern>{YOUR NAME}</Pattern>
  <Response>My name is <Bot Get="Name" /></Response>
</Model>
```

➔ Type "**What is your name?**" in the Console
➔ The Bot will reply "***My name is Syn Web Assistant***"

Keywords

Sometimes its very convenient to use keywords to declare a pattern. Keywords based pattern saves a lot of time but has a penalty of being extremely ambiguous. Keywords in SIML (within the pattern tag) should be enclosed within curly brackets.

Note:

A keyword based pattern will match if the the sequence of words occurs within the user input even if multiple words exist before, between or even after the keywords declared.

❊ Values for the Get attribute for a Bot element are case-insensitive

_ Underscore wildcard

➔ Type the following SIML Code within "**My First Concept**"

SIML Code

```
<Model>
  <Pattern>WHAT IS THE MEANING OF _</Pattern>
  <Response>
    <Goto>Define <Match /></Goto>
  </Response>
</Model>

<Model>
  <Pattern>WHAT IS THE MEANING OF *</Pattern>
  <Response>I don't really know.</Response>
</Model>

<Model>
  <Pattern>DEFINE LIFE</Pattern>
  <Response>42</Response>
</Model>
```

❊ Patterns in SIML are case-insensitive

❊ From this point on we will no longer have sections showing the entire SIML code for readability

_ Underscore wildcard explained

➔ Type "**What is the meaning of Life?**" in Console
➔ The Bot replies "***42***"

Underscore wildcard and priority

The _ wildcard has a priority higher than the * wildcard and therefore the Model with the _ wildcard is activated instead of the Model with the * wildcard.

Once the Model is activated the **<Goto>** tag is used to redirect the pattern search to simpler pattern i.e. "**Define**" followed by whatever is matched by the _ wildcard. So the new search becomes "**Define Life**". The "*Pattern Reductions*" file under the "**Settings**" category in your SIML project is to group multiple pattern reductions just like the one you wrote previously.

SIML Code

```
<Goto>Define <Match /></Goto>
```

❊ Pattern Reductions file should be used to store all pattern redirections that involve <Goto> or <GotoMatch/> elements

SIML Pattern elements

Priority of Pattern elements in SIML

^ (Up arrow) [Rank 1]

Has the highest priority

If (Conditional) [Rank 2]

A Conditional Pattern in SIML ranks second in the pattern search

Js (Scripted) [Rank 3]

A Scripted pattern is evaluated after the search for a conditional pattern ends

% (Percent) [Rank 4]

Matches zero or more words

_ (Underscore) [Rank 5]

Matches one or more words

Word [Rank 6]

A word in any language within the pattern tag

[Set] [Rank 7]

A collection of words grouped under a single name

(Word1|Word2) [Rank 7]

A Dynamic set has the same rank as a normal Set and are treated as such

@Regex [Rank 8]

Regular Expressions are evaluated after Sets

$ (Dollar Sign) [Rank 9]

Matches zero or more words but has a lower priority than **%**

{Word1 Word 2} [Rank 10]

Keywords are evaluated just before the final wildcard *

*** (Asterisk) [Rank 11]**

Matches one or more words and has the least priority

❊ Not all pattern types have been discussed in this presentation

Regular Expressions in patterns

➔ Click on “Regular Expressions” under “Settings” category and type the following within the SIML element.

SIML Code

```
<Regex Name="switch" Pattern="\b(on|off)\b"/>
```

➔ Now click on “My First Concept” file under “Files” category and type the following SIML Code within the Concept
➔ Press F5 to refresh the Project

SIML Code

```
<Model>
  <Pattern>Switch it @switch</Pattern>
  <Response>Switched <Match /></Response>
</Model>
```

❊ The name of a Regex should be unique

Test your Regular Expression

- ➔ Type “Switch it off” in the Console and press Send
- ➔ The Bot replies “Switched off”
- ➔ Type “Switch it On” in the Console
- ➔ The Bot replies “Switched On”

Regular Expressions

Regex in SIML requires a Name and ofcourse a regex pattern. SIML Bot internally optimizes your regex to work at peak performance during a chat session.

SIML Code

```
<Regex Name="switch" Pattern="\b(on|off)\b"/>
```

In the above code the name “**Switch**” is used to give a regex pattern a unique name and the regex pattern itself is declared as a value for the Pattern attribute

A declared regular expression can be used within a pattern by prefixing the name of the regular expression (set by you) with an at symbol. Example **@Switch**

❊ Regex names are case-insensitive

Sets in patterns

➔ Click on “Sets” under “Settings” category and type the following within the SIML tag

SIML Code

```
<Set Name="Color">
  <Item>Red</Item>
  <Item>Green</Item>
  <Item>Blue</Item>
</Set>
```

➔ Now click on “My First Concept” file under “Files” category and type the following SIML Code within the Concept
➔ Press F5 to refresh the Project

SIML Code

```
<Model>
  <Pattern>I like the color [Color]</Pattern>
  <Response>I like <Match /> too</Response>
</Model>
```

❊ Press Alt+T to insert an <Item> tag
❊ Declared Sets should always be stored in the Sets file under the Settings category

Test your Set

- ➔ Type “I like the color red” in the Console and press Send
- ➔ The Bot replies “I like red too”
- ➔ Type “I like the color BLUE” in the Console
- ➔ The Bot replies “I like BLUE too”

Sets

A Set in SIML is a collection of simple words. Words that group entities/nouns. Example: a Set **Weekday** will contain Sunday, Monday, Tuesday and so on. Large sets have no impact on SIML processing speed and are limited by your computer’s memory (RAM).

To declare a set you should give the Set a unique name using the Name attribute. Set items are stored within the <Item> tag

SIML Code

```
<Set Name="Color">
 <Item>Red</Item>
 <Item>Green</Item>
 <Item>Blue</Item>
</Set>
```

Greedy

SIML sets are greedy i.e. SIML Bot tries to find the best match in a set

❊ Set names are case-insensitive.

Nested Modelling

Models within Models

In SIML Models can occur with other Models. The behaviour of such a nested structure is as if the Models were connected using independent <*Previous*> elements that had the Response of the Parent Model as its value.

SIML Code

```
<Model>
   <Pattern>START APPLICATION</Pattern>
   <Response>Are you sure ?</Response>
   <Model>
     <Pattern>Yes</Pattern>
     <Response>Application started.</Response>
   </Model>

   <Model>
     <Pattern>No</Pattern>
     <Response>As you wish.</Response>
   </Model>
</Model>
```

Add the above SIML code to your current Concept in your Project.

Test your Nested Models

- ➔ Type “Start application” in the Console and press Send
- ➔ The Bot replies “Are you sure”
- ➔ Type “Yes” in the Console
- ➔ The Bot replies “Application Started”
- ➔ Repeat the steps but this time say “No”
- ➔ And the Bot will reply “As you wish.”

Behaviour

Models within Models is a special technique to avoid writing multiple **<Previous>** elements whenever you wish to refer to the last output of the Bot. Nested Models do not take the previous utterance of the Bot into consideration instead they check if the Parent Model was activated by the previous user input.

Nested Modelling should be used if the response generated by the Parent Model does not belong to a general Concept.

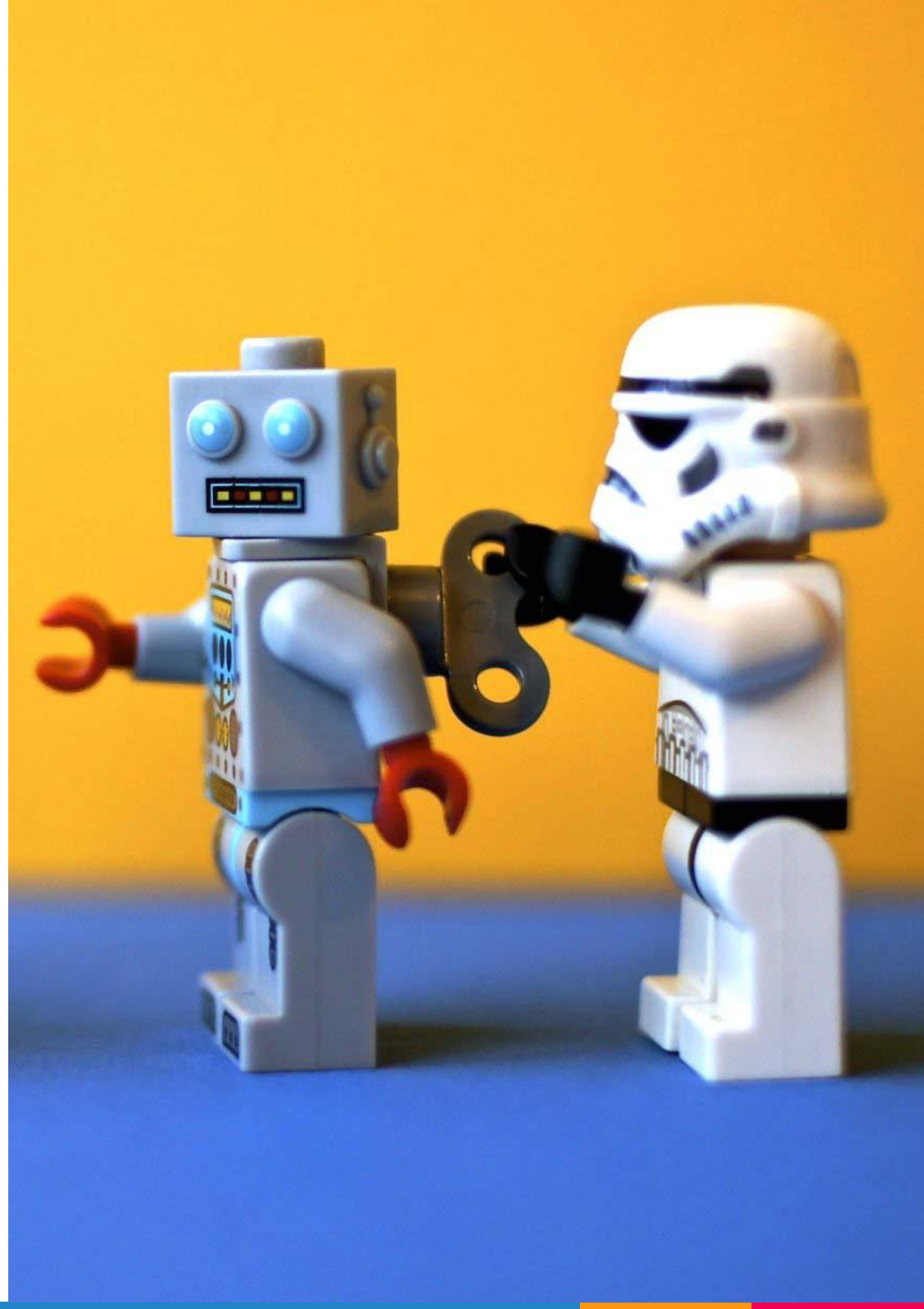
Nested Models also eliminate the requirement of a **<Label>** element if the Parent Model generates a random response.

❊ More information is available in Wiki

Settings

A Setting in SIML is a collection of Variables that are stored in relation to its owner (Bot or User)

Some variable names are reserved by default and are used to manipulate the behaviour an SIML code



Bot Settings

Variables of a Bot

➔ Click on “Bot-Settings” under the Settings category in Files Explorer

BotSettings

The *BotSettings* element acts as a container for all Bot related variables (or predicates). A Variable in SIML should have a name and can have 1 or more values. Some variables can be left “**undefined**” meaning that they exists but have no values in them.

➔ Type in the following Code within the BotSettings element

SIML Code

```
<Variable Name="sleep-time" Value="11pm"/>
```

❊ Variable names should be unique and names that require 2 or more words should be separated by a hyphen.

Bot Settings

Getting a Variable

➔ Click on “My First Concept” and type the following SIML Code

SIML Code

```
<Model>
  <Pattern>WHEN DO YOU SLEEP</Pattern>
  <Response>I sleep at <Bot Get="sleep-time" /></Response>
</Model>
```

➔ Type “when do you sleep” in Console
➔ The Bot replies “I sleep at 11pm”

Get

The Get attribute takes a variable name as its value and returns the value of the variable specified. A variable name is case-insensitive in SIML but by convention use - (hyphen) to separate individual words in a variable name.

❊ Always remember to Press F5 to refresh your project once you’ve made changes to any SIML document

Bot Settings

Setting a Variable

➔ Replace the previous SIML code with the following

SIML Code

```
<Model>
 <Pattern>WHEN DO YOU SLEEP</Pattern>
 <Response>I sleep at <Bot Get="sleep-time" />
   <Think>
     <Bot Set="sleep-time">12am</Bot>
   </Think>
 </Response>
</Model>
```

➔ Type “when do you sleep” in Console
➔ Bot Replies “I sleep at 11pm”
➔ Type again “When do you sleep” in Console
➔ Bot Replies “I sleep at 12am”

Set

The Set attribute is used to set a value for a variable (that may or may not be defined). The **<Think>** tag in the code above is used to silence the output by the Bot element which would otherwise return 12am

User Settings

Variables of a User

➔ Click on “User-Settings” under the Settings category in Files Explorer

UserSettings

Just like the *BotSettings* tag the **UserSettings** acts as a container for all **User** related variables. All values defined within the User-Settings files are considered default for all newly created users. And help the bot to assume some information regarding the user it may interact with.

- Just like BotSettings you can use variables within UserSettings

SIML Code (Example)

```
<Variable Name="Middle-Name" Value="Edward"/>
```

SIML Code (Example)

```
<Variable Name="favorite-movies">
  <Value>Contact</Value>
  <Value>Godfather</Value>
  <Value>Green Miles</Value>
  <Value>Life Is Beautiful</Value>
</Variable>
```

User Settings

Getting a Variable

➔ Click on “My First Concept” and type the following SIML Code

SIML Code

```
<Model>
  <Pattern>What is my middle name</Pattern>
  <Response>It’s <User Get="middle-name" /></Response>
</Model>
```

➔ Type “what is my middle name” in Console
➔ The Bot replies “It’s Edward”

Setting a Variable

Like the Bot tag the User tag can be used to set a value for a user variable using the **Set** attribute whose value should be a variable name.

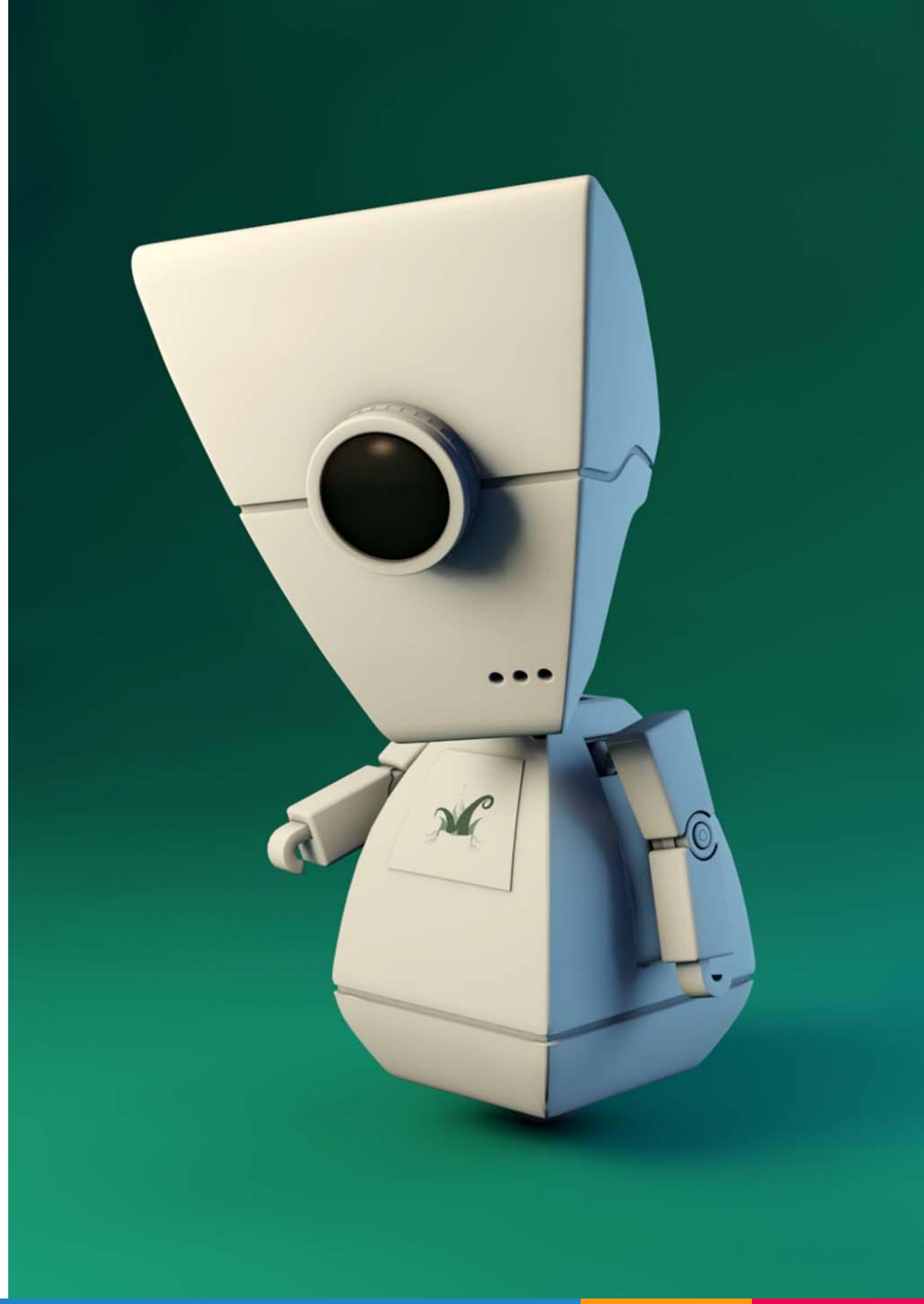
SIML Code

```
<Model>
  <Pattern>My name is *</Pattern>
  <Response>Hi there! <User Set="Name"><Match /></User></Response>
</Model>
```

❊ Both Bot and User tags have the same set of operations

Normalization

Normalization in SIML involves filtration of unimportant words or symbols, replacement of words and splitting of user message into multiple Inputs



Normalization

SIML Normalization elements

Filter

A Filter in SIML is used to replace characters or words in user input or output

Filters are of 3 types

- Input
- Output
- Both

Splitter

A Splitter in SIML is used to split the user message into multiple sentences wherever certain characters are found.

<Text>, <Word> and <Regex>

These 3 tags can be used within a Filter or a Splitter element

- <Text> denotes a sequence of characters
- <Word> denotes an actual word
- <Regex> a regular expressions that match complex sequence of characters

Filter

Creating a very simple Filter

➔ Click on “Normalizations” under the Settings category and type the following within the SIML element

SIML Code

```
<Filter Value="bla">
 <Word>bla bla</Word>
</Filter>
```

In the code above we create a *Filter* by using the **<Filter>** tag and add a **<Word>** tag with the value bla bla. The Value attribute in the above filter will be used as the replacement text that will replace all occurrences of the words “**bla bla**” with just “**bla**”

Complexity

Filtration is not a simple task and requires planning. Planning of filtration is required in complex projects because the developer has to take into consideration the usage of Regular Expressions and has to make sure certain characters or words are not filtered out.

Test your Filter

➔ Click on “My First Concept” file under Files and type the following SIML Code

SIML Code

```
<Model>
  <Pattern>BLA</Pattern>
  <Response>What!</Response>
</Model>
```

➔ Type “bla bla” in Console
➔ The Bot replies “What!”

As filtration takes place before an Input is processed your input “**bla bla**” is filtered out and is replaced with “**bla**” and that is why the above Model gets activated.

Types

As the Filter you defined previously has no Target attribute it is assumed to be an Input filter. Input filters only replace user inputs. There is also an option to create an Output filter which filters out Bot messages.

Splitter

A Splitter simply splits the user message into multiple sentence wherever the specified character, word or even a regex is matched.

Typical English Splitter

We recommend using just . (dot) and ?(question mark) as the 2 textual characters in Splitters if you are developing an English/French/Spanish/ Bot.

SIML Code

```
<Splitter>
  <Text>.</Text>
  <Text>?</Text>
  <Text>!</Text>
  <Text>;</Text>
</Splitter>
```

- The above Splitter code is already present within the SIML element in the file “Normalizations”. You may notice that we have used <Text> tags above and this is to tell the SIML bot that wherever the characters (not words) are found the sentence should be split.

Test sentence splitting

➔ Click on “My First Concept” file under Files and type the following SIML Code

SIML Code

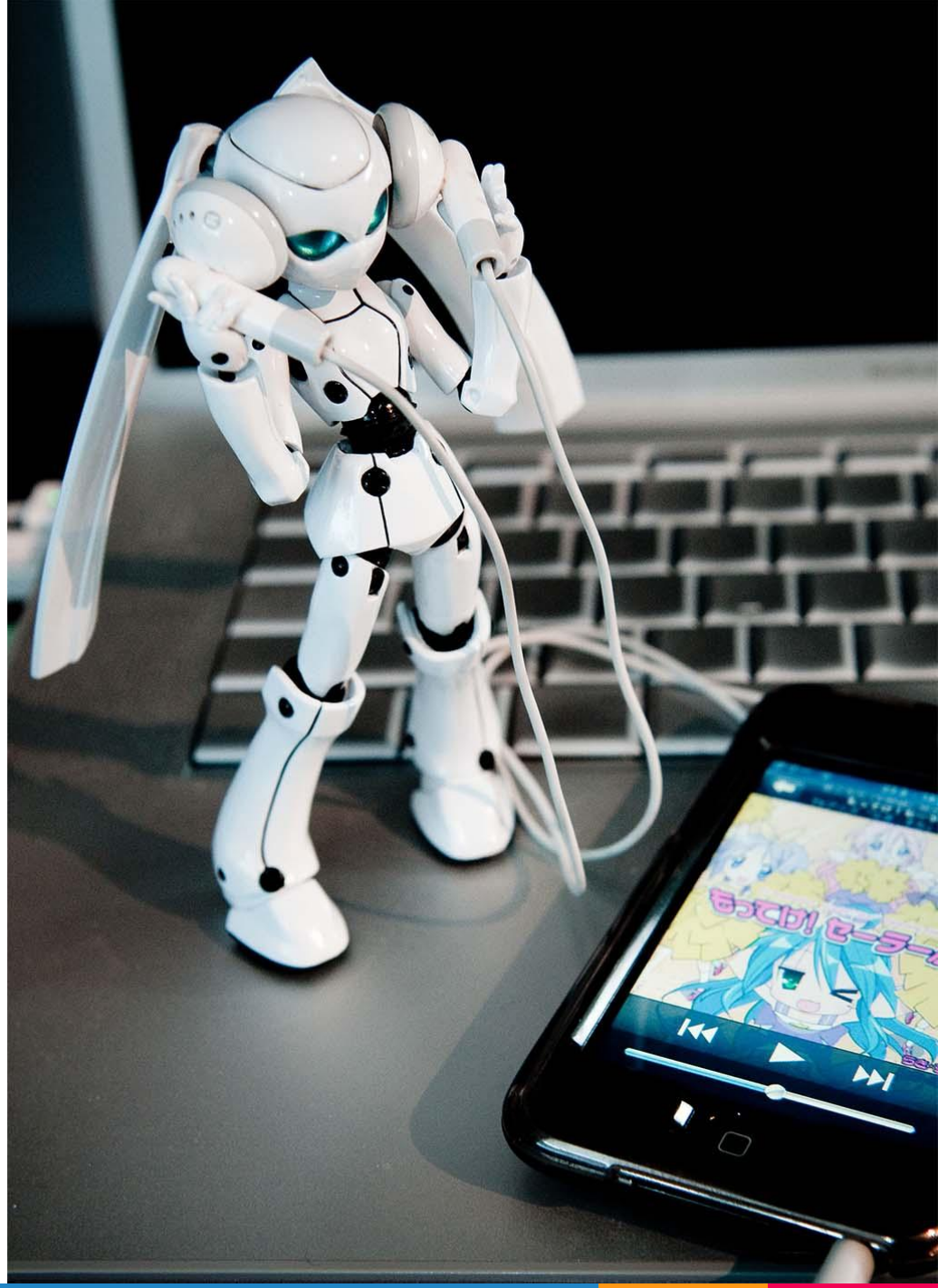
```
<Model>
  <Pattern>HOW ARE YOU</Pattern>
  <Response>I am fine thank you.</Response>
</Model>

<Model>
  <Pattern>ANYTHING NEW</Pattern>
  <Response>Yes a lot has changed now.</Response>
</Model>
```

➔ Type “how are you? anything new?” in Console
➔ The Bot replies “I am fine thank you. Yes a lot has changed now.”

- The user input is split into “**how are you**” and “**anything new**” because of the character “?” which has already been declared as the splitting character within the Splitter element discussed earlier.

Randomness

Random responses make your Bot feel more humanely by generating unique random outputs for the same set of pattern.



Random Response

There are 2 types of Random response elements in SIML

Random

Under this element individual responses are saved using the <Item> tag and whenever the Random element is evaluated after a Model is activated the Random element returns a random item from the list within it.

Phrase

Similar to a Random element but instead of just selecting a random element the Phrase element selects the item within its list that best matches the user input.

But in some cases the Phrase element will return a Random value if the “Word Distance” is approximately the same for all listed items.

Creating a Random Response

➔ Replace previous Model with the pattern “how are you” with the following SIML Code

SIML Code

```
<Model>
   <Pattern>HOW ARE YOU</Pattern>
   <Response>
     <Random>
       <Item>I am doing great</Item>
       <Item>I am fine thank you.</Item>
       <Item>Never been better</Item>
       <Item>Fine. How are you?</Item>
     </Random>
   </Response>
 </Model>
```

➔ Type “**How are you?**” in the Console
➔ A random answer is generated
➔ Type “**How are you?**” again in the Console
➔ A random answer is generated

❊ Pressing Alt+R inserts a Random element template

Creating a Predictive Response

➔ Type in the following SIML Code

SIML Code

```
<Model>
  <Pattern>
    <Item>WHAT IS YOUR NAME</Item>
    <Item>WHAT CAN I CALL YOU</Item>
  </Pattern>
  <Response>
    <Phrase>
      <Item>My name is <Bot Get="Name" /></Item>
      <Item>You can call me <Bot Get="Name" /></Item>
    </Phrase>
  </Response>
</Model>
```

➔ Type "**What is your name?**" in the Console
➔ Bot replies "***My name is Syn Web Assistant***"
➔ Type "**What can I call you?**" in the Console
➔ Bot replies "***You can call me Syn Web Assistant***"

❊ Press Alt+P to insert a Phrase element template
❊ Multiple patterns can be grouped within the same Model using the <Item> tag

Reusability - Random Response

Random Responses file

Under the Settings category you will see the “Random Responses” file. This file is used to save Random responses for later usage. To save a Random/Phrase element for later reuse add a Random element and give it a unique name for identification using the “Name” attribute. Later on you can reuse the Random element using the “Get” attribute.

Type the following Code within the “Random Responses” file

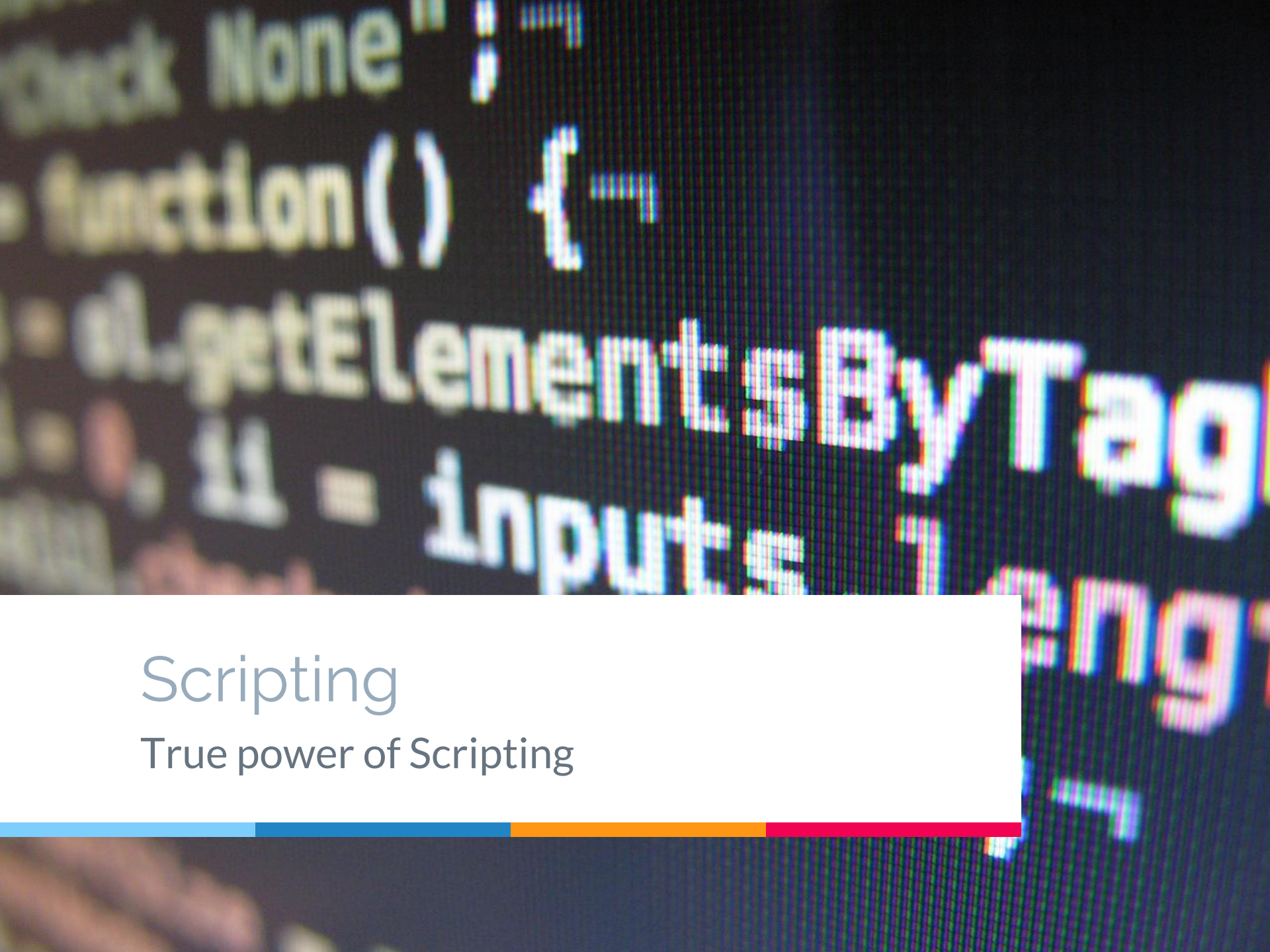
SIML Code

```
<Random Name="I-am-fine">
  <Item>I am doing great</Item>
  <Item>I am fine thank you.</Item>
  <Item>Never been better</Item>
  <Item>Fine. How are you?</Item>
</Random>
```

Replace your Random element in the previous Random example with the Code below and the Bot’s response should remain the same.

SIML Code

```
<Random Get="i-am-fine" />
```



Scripting

True power of Scripting

Scripts in SIML

SIML gives Scripts the true power

Language

Scripts in SIML 1.0 are written using JavaScript as SIML Bot has its own JavaScript Interpreter that complies with ECMA-5.1. You may use a script within the following SIML tags.

Within <Pattern>

A Script can be used within a Pattern and if the result of evaluation returns True the Model is activated

Within <Previous>

Scripts inside a Previous element are evaluated even before a Pattern is evaluated

Within <Response>

Scripts inside a Response tag can be used to generate complex responses.

- If you are planning to use Scripts in Patterns we highly recommend using them inside either a Concept that is declared *private* or within a Model that contains a *Previous* tag

❊ Scripts are usually slower when compared to SIML elements and therefore should be used when absolutely necessary

Writing your first Script in SIML

- ➔ Click on the file "**Scripts**" under the *Settings* category"
- ➔ Replace the example script with the following

SIML Code

```
<Script Type="javascript">
 function helloScript(){
    return "Hello from Js!";
  }
</Script>
```

We have now created a "**Global**" JavaScript that can be called at runtime. The above script is extremely simple in nature and you would probably use a much more complex function while developing an SIML Bot. All global scripts should be stored within this "Scripts" file. The Type attribute for SIML 1.0 should always be set to "JavaScript".

We will now test our JavaScript inside an SIML response.

❊ Signatures of a global Scripts are evaluated once during startup and can be reused during a Chat session

Calling a Script

➔ Click on the file "**My First Concept**" under the *Files* category
➔ And type the following SIML Code

SIML Code

```
<Model>
  <Pattern>HELLO JAVASCRIPT</Pattern>
  <Response>
     <Js>helloScript();</Js>
  </Response>
</Model>
```

➔ Type "Hello javascript" in Console
➔ Bot replies "Hello from Js!"

Your *globally* declared function can now be reused whenever required. Additionally you could also type the globally declared JavaScript within the Js element and the response would still be the same.

For more details on how Scripts can be used within Patterns and Responses refer Wiki

❊ Js tag is used to execute a JavaScript

Maps

Transforms a collection of words that belongs to one set into an element of another

Maps

➔ Click on the file "**Maps**" under the *Settings* category
➔ And add the following SIML Code

SIML Code

```
<Map Name="person2">
  <MapItem Content="like you" Value="like me" />
</Map>
```

➔ Now add the following SIML Code within "**My First Concept**"

SIML Code

```
<Model>
   <Pattern>I LIKE CHATBOTS *</Pattern>
    <Response>
       That's great that you like chatbots <Map Get="person2"><Match /></Map>
    </Response>
</Model>
```

➔ Type "**I like chatbots like you**" in Console
➔ The Bot replies "***Thats great that you like Chatbots like me***"

Maps

Usage

Maps comes in handy when textual transformations are required. Example a Map **Number2Word** may transform 1, 2, 3 and so on to One, Two, Three. They can also used for pronoun substitutions say first-person to second-person or second-person to first-person.

Default **English** Project template in Syn Chatbot Studio comes with many regularly used Maps. To get a Map use the **Map** tag and specify the name of the map using the **Get** attribute.

SIML Code (Example)

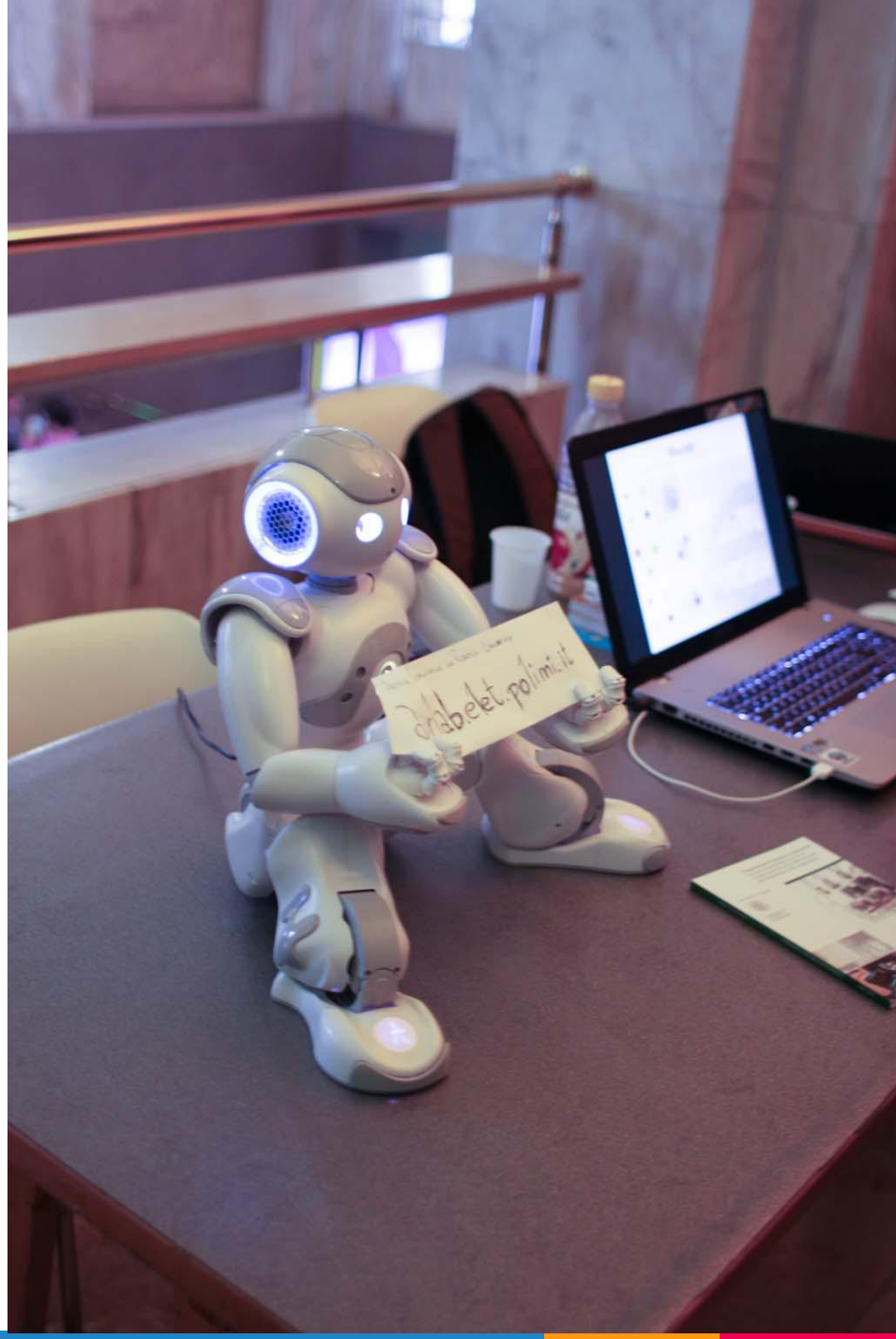
```
<Map Get="person2"><Match /></Map>
```

Map operations are extremely fast and larger Maps have no impact on the performance of the Bot

❊ Map names are case-insensitive

Repetition

SIML's Repeat Management allows developers to manage repeated user inputs within Non-Repeatable Concepts



Repeat Tag

Repetition

In SIML a repetition is not what the user repeats but its what the Bot has to repeat to answer a chat request. This is because there are number of ways in SIML to get a Model activated and a lot of sentences may hold the same meaning. SIML addresses repetitive user requests using a unique mechanism where the Bot checks what the Bot had to repeat, if the repeated output belonged to a Non-Repeatable Model and how many of the Non-Repeatable Models were activated.

Repetition Management is an advanced topic and therefore we encourage you to refer the Wiki before diving deep into it. We will now look at an example SIML code that handles repetition using the **<Repeat>** and **<Echo>** tags.

❊ An SIML Project can have just one <Repeat> element. Multiple occurrences will override existing mechanisms

Repetition Management

SIML Code

```
   <Repeat>
    <Response>
      <If User="partial-repeat">
        <Switch User="repeat-count">
          <Case Value="1">And I have already mentioned <Echo Index="1" /></Case>
          <Case Value="2">And I think you already know <Echo Index="1" /> and <Echo
Index="2" /></Case>
          <Case Value="3">And If you recall our conversation you would know <Echo
Index="1" />, <Echo Index="2" /> and <Echo Index="3" /></Case>
          <Default>And I do not like to repeat myself.</Default>
        </Switch>
      </If>
      <Else>
        <Switch User="repeat-count">
          <Case Value="1">I have already mentioned <Echo Index="1" /></Case>
          <Case Value="2">I think you already know <Echo Index="1" /> and <Echo
Index="2" /></Case>
          <Case Value="3">If you recall our conversation you would know <Echo
Index="1" />, <Echo Index="2" /> and <Echo Index="3" /></Case>
          <Default>I do not like to repeat myself.</Default>
        </Switch>
      </Else>
    </Response>
  </Repeat>
```

❊ This is a template SIML code and is available by default in all SIML projects

Repetition

<Repeat> and <Echo>

In the previous code we check if part of the user's input activated some (not all) Non-Repeatable Models and if it did then we later check how many of the Non-Repeatable Models were activated.

Using the **<Echo>** tag we tell the user what was previously mentioned (this Echo may not be the exact output the Bot generated last time as its a fresh response from a Non-Repeatable Model).

If you are unsure as to how repetitions are Managed in SIML we recommend leaving the Code as such if you are developing an English Bot.

❊ A few SIML elements are Not valid within a Repeat element

Conclusion

We have barely scratched the surface with this presentation and we highly recommend that you go through SIML Wiki to create intelligent Bots.

Wrapping it all

Developer Network

Syn Developer Network elaborates all of the concepts discussed in this documentation. So if you need more details you know exactly where to look for.

Studio

Syn Bot Studio is the official IDE and the main playground for working with SIML projects.

Developer Forum

If you have any questions or suggestions Syn is open to you. Unleash the Synner in you and get together with us at Forum.Syn.co.in

SIML

SIML will always be improved and any specification changes will not hamper your development as our IDE will automatically upgrade the projects for you.

AI Freedom

All SIML knowledge base templates are free for both Commercial and noncommercial applications. To give the users the true Power SIML files are licensed under Apache License Version 2.0

0 Redundancy

With SIML we aim to eliminate the decade old code redundancy using automated tools and analysis.

Presentation Version 1.0 2014-12

Products and Specifications used in this documentation

- Specification: **SIML 1.0**
- Software: **Syn Bot Studio 1.0 Beta**

You can download all SIML Projects for Free and make changes:

https://github.com/SynHub

Licenses involved with SIML and its derivatives

- **SIML Files -** Apache License Version 2.0
- **SIML Bot 1.0 [C# Portable] -** Syn Freeware License
- **Syn Bot Studio 1.0 Beta -** Syn Freeware License

Thanks!

Any questions?

Visit: Forum.Syn.co.in

Synthetic Intelligence Network