=========================================

코드 검토 가이드 라인 및 모범 사례

=========================================

SmartApp 또는 Device Handler를 제출하기 전에 본인의 코드가 이 문서에 기재 된 지침을 준수하는지 확인해야합니다.

이 가이드 라인을 준수하지 않는 코드는 거절될 수 있습니다.

이 문서는 SmartThings 개발을 위한 모범 사례 모음집이기도 합니다.

----

일반적인 규칙

-------

코드는 읽기 쉬워야합니다.

^^^^^^^^^^^^^^^^^^^^^^^

코드는 기계가 실행하지만, 사람이 읽습니다.

가독성은 주관적 일 수 있지만 따라야 할 몇 가지 일반적인 지침이 있습니다:

- 의미 있는 변수명 및 메소드 이름을 사용하세요.

- :참고:`review\_guidelines\_dry`

- :참고:`review\_guidelines\_methods`

- :참고:`review\_guidelines\_comments`

.. \_review\_guidelines\_dry:

중복하지 마세요

^^^^^^^^^^^^^^^^^^^^^

`DRY 법칙 <https://en.wikipedia.org/wiki/Don%27t\_repeat\_yourself>`\_\_ (don't repeat yourself 중복하지 말라)을 따라주세요.

코드를 복사/붙여넣기 하지 마세요 – 자주 쓰이는 코드를 공유 유틸리티 메소드로 빼주세요.

.. \_review\_guidelines\_methods:

메소드는 한가지 목적만 가져야합니다

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

메소드는 한가지 목적만을 충족시켜야하며 간결해야합니다.

메서드의 정의가 표준 컴퓨터 화면을 넘어간다면, 너무 길다는 겁니다.

코드를 유틸리티 메서드로 분리할 수 있는지 알아보세요.

예를 들어, 큰 사이즈의 HTTP 응답을 즉시마다 분석하는 메서드는 길어질 수 있으니, 이 작업을 호출할 수 있는 여러 메서드로 분리시키세요.

이렇게 하면, 코드를 더 쉽게 이해할 수 있게 되며 더 나은 `관심사의 분리 <https://en.wikipedia.org/wiki/Separation\_of\_concerns>`\_\_를 보장합니다.

사용하지 않는 코드를 제출하지 마세요

^^^^^^^^^^^^^^^^^^^^^^^^^

사용하지 않거나 주석처리 된 코드는 제출하기 전에 지워 주셔야합니다.

모욕적인 말, 모독적인 언어 또는 비방하는 언어를 사용하지 마세요

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

따로 설명이 필요 없겠지만 – 언어는 깔끔하고 전문적이어야합니다.

.. \_review\_guidelines\_comments:

Comment appropriately

^^^^^^^^^^^^^^^^^^^^^

Comments can add clarity and context to code when used appropriately.

When over-used, they clutter the code and provide no value.

There are some guidelines that should be followed:

- In general, when the code is doing something out of the ordinary, a comment is appropriate.

- Device Handler custom commands and attributes should have a comment describing the purpose, parameters, and exception conditions (if applicable).

- Non-trivial methods should be documented with comments describing what it does, its return type, exception conditions, and parameters. `JavaDoc style comments <https://en.wikipedia.org/wiki/Javadoc#Overview\_of\_Javadoc>`\_\_ can be used, though there is no tooling in place to generate documentation from the source.

- Comments should add value - commenting every line of readable code simply clutters the code and is unnecessary.

Here's an example of using comments appropriately for documenting a method:

.. code-block:: groovy

def capabilityCommands = getDeviceCapabilityCommands(device.capabilities)

/\*\*

\* Builds a map of capability names to their supported commands.

\*

\* @param a list of Capabilities.

\* @return a map of device capability -> supported commands.

\*/

def getDeviceCapabilityCommands(deviceCapabilities) {

def map = [:]

deviceCapabilities.collect {

map[it.name] = it.commands.collect{ it.name.toString() }

}

return map

}

Here's an example of an in-line code comment explaining why the code is checking if a percentage value is within a certain hard-coded range:

.. code-block:: groovy

log.trace "stopDimmersHandler evt: ${evt.value}"

def percentComplete = completionPercentage()

// Oftentimes, the first thing we do is turn lights on or off,

// so make sure we don't stop as soon as we start

if (percentComplete > 2 && percentComplete < 98) {

...

}

An example of inappropriate comments is below.

Note how the comments simply repeat what is obvious by reading the code; no value is added.

.. code-block:: groovy

// get all the children

def children = pollChildren()

// iterate over all the children

children.each {child ->

// log each child

log.debug "child: $child"

}

Handle all ``if()`` and ``switch()`` cases

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Make sure any ``if()`` or ``switch()`` blocks handle all expected inputs.

Forgetting to handle a certain condition can cause unexpected logic errors.

Also, every ``switch()`` statement should have a ``default:`` case statement to handle any cases where there is no match.

Verify assumptions

^^^^^^^^^^^^^^^^^^

If a method operates on some input, it should handle all possible input values, including any differences if the method is called from a parent or child SmartApp or Device Handler.

Use consistent return values

^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Groovy is a dynamically typed language.

That's great for a lot of things, but it's a sharp knife - highly effective, yet also easy to cut yourself accidentally.

A method should return a single type of data, regardless of if the method signature is typed or not.

For example, don't do something like this:

.. code-block:: groovy

def getSomeResult(input) {

if (input == "option1") {

return true

}

if (input == "option2") {

return false

}

return [name: "someAttribute", value: input]

}

The example above fails to return a consistent data type.

Calling clients of this code have to accommodate both a boolean and map return values.

Instead, methods should always return the same data type.

.. note::

In certain cases, it \*may\* make sense for a method to return different types.

Such cases are the exception, and the different types returned, and under what circumstances, should be documented in the method's comments.

Be careful indexing into arrays

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

When parsing data, pay attention to arrays if you use them.

Do not index into arrays directly without making sure that the array actually has enough elements.

Consider the following code that splits a string on the ``":"`` character, and returns the value after the ``":"``:

.. code-block:: groovy

def getSplitString(input) {

return input.split(":")[1]

}

// -> "123"

getSplitString("abc:123")

// -> ArrayIndexOutOfBounds exception!

getSplitString("abc:")

Because ``getSplitString()`` does not verify that the result of ``split()`` split has more than one element, we get an ``ArrayIndexOutOfBounds`` exception when trying to access the second item in the parsed result.

In cases like this, make sure your code verifies the array contains the item:

.. code-block:: groovy

def getSplitString(input) {

def splitted = input?.split(":")

if (splitted?.size() == 2) {

return splitted[1]

} else {

return null

}

}

Use the Elvis operator correctly

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Groovy supports the Elvis operator, which allows us write more concise conditional expressions than otherwise possible.

However, we need to understand :ref:`Groovy truth <review\_guidelines\_groovy\_truth>` to use it effectively.

Consider this example that attempts to set the variable ``bulbLevel`` to ``100`` if it is not already set:

.. code-block:: groovy

def bulbLevel = settings.level ?: 100

But what happens if ``settings.level`` is ``0`` in the example above? \*\*Because Groovy considers zero as false, we've set\*\* ``bulbLevel`` \*\*to\*\* ``100`` \*\*!\*\*

The above expression should be rewritten as:

.. code-block:: groovy

def bulbLevel = settings.level == null ?: 100

Handle null values

^^^^^^^^^^^^^^^^^^

.. important::

NullPointerExceptions are one of the most frequently occurring exceptions on the SmartThings platform - take care to avoid them!

This is \*very\* common in LAN and SSDP interactions, so always double check that code.

A ``NullPointerException`` will terminate the SmartApp or Device Handler execution, but can be avoided easily with the `safe navigation <http://groovy-lang.org/operators.html#\_safe\_navigation\_operator>`\_\_ (``?``) operator.

Any code that may encounter a ``null`` value should anticipate and handle this.

The examples below show a few common scenarios in which ``null`` is possible, and how to deal with it using the ``?`` operator:

.. code-block:: groovy

// if the LAN event does not have headers, or a "content-type" header,

// don't blow up with a NullPointerException!

if (lanEvent.headers?."content-type"?.contains("xml")) { ... }

.. code-block:: groovy

// if a location does not have any modes, statement simply returns null

// but does not throw a NullPointerException

if (location.modes?.find{it.name == newMode}) { ... }

.. \_review\_guidelines\_groovy\_truth:

Use Groovy truth correctly

^^^^^^^^^^^^^^^^^^^^^^^^^^

Be aware of, and ensure your code is consistent with, what Groovy considers true and false.

Groovy truth is documented `here <http://groovy-lang.org/semantics.html#Groovy-Truth>`\_\_.

Here are some gotchas to be aware of:

- Empty strings are considered ``false``; non-empty strings are considered ``true``.

- Empty maps and lists are considered ``false``; non-empty maps and lists are considered ``true``.

- Zero is considered ``false``; non-zero numbers are considered ``true``.

Consider the following example that verifies that a number is between 0 and 100:

.. code-block:: groovy

def verifyLevel(level) {

if (!level) {

return false

} else {

return (level >= 0 && level <= 100)

}

}

If we call ``verifyLevel(0)``, the result is ``false``, because ``0`` is treated as false by Groovy.

Instead, it should be written as:

.. code-block:: groovy

def verifyLevel(level) {

return (level instanceof Number && level >= 0 && level <= 100)

}

This can be a common source of errors; make sure you understand and use Groovy truth appropriately.

----

Using State

-----------

``state`` is not an unbounded database

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

``state`` (SmartApps and Device Handlers) and ``atomicState`` (SmartApps only) are provided to persist small amounts of data across executions.

Do not think of state as a virtually unlimited database for your app.

The amount of data that can be stored in state is :ref:`limited <state\_size\_limit>`.

Avoid code that adds items to ``state`` regularly (perhaps in response to Events or schedules), but does not remove items.

Understand how ``state`` works

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Remember that when using ``state``, the :ref:`results are not persisted until the app is done executing <state\_how\_it\_works>`.

This can have unintended consequences, such as state values being overridden by another concurrently executing instance of the SmartApp.

Understand when to use ``atomicState`` vs. ``state``

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Understand the :ref:`difference <choosing\_between\_state\_atomicState>` between ``atomicState`` and ``state``, make sure you use the correct one for your needs, and avoid using both in the same SmartApp.

Take care when storing collections in ``atomicState``

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Modifying collections in Atomic State does not work as it does with State.

:ref:`Read the documentation <atomic\_state\_collections>` to understand how to best work with collections stored in Atomic State.

----

Web Services

------------

Document external HTTP requests

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

:ref:`HTTP requests <calling\_web\_services>` to outside services should be documented, explaining the need to make external requests, what data is sent, and how it will be used.

Please also include a comment with a link to the third party's privacy policy, if applicable.

Document any exposed endpoints

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

If your SmartApp or Device Handler :ref:`exposes any endpoints <web\_services\_mapping\_endpoints>`, add comments that document what the API will be used for, what data may be accessed by those APIs, and where possible, include a link to the privacy policies of any remote services that may access those APIs.

----

Scheduling

----------

Avoid recurring short schedules

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Scheduled and other periodic functions should not execute more often than every five minutes, unless there is a good reason for it, and the reviewers agree.

If your code executes more frequently than every five minutes, add a comment to your code explaining why this is necessary.

Avoid chained ``runIn()`` calls

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

:ref:`Do not chain runIn() calls<scheduling\_chained\_run\_in>`.

If for some reason it is necessary, add a comment describing why it is necessary.

----

Security considerations

-----------------------

Subscriptions should be clear

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

It is possible to subscribe to Events using a string variable, so what the SmartApp is subscribing to might be somewhat opaque.

For example:

.. code-block:: groovy

def myContactSubscription = "contact.open"

...

subscribe(contact1, myContactSubscription, myContactHandler)

The best practice is to subscribe explicitly to the attribute:

.. code-block:: groovy

subscribe(contact1, "contact.open", myContactHandler)

However, if the SmartApp must subscribe to a variable (from state, for instance), the reviewer should be able to trace how the variable is set and what the expected attribute will be.

Subscriptions should be specific

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Do not create overly-broad subscriptions.

A SmartApp that is subscribed to every location Event will execute excessively, and is rarely necessary.

Instead, create subscriptions specific to the Event you are interested in.

If you're creating a service manager for a LAN-connected device, be sure to :ref:`subscribe to the device search target <lan\_device\_discovery>`.

Do not use dynamic method execution

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

In groovy you can execute functions based on a string, like so:

.. code-block:: groovy

object."${mystring}"()

Which can be very handy, but when ``${mystring}`` comes from a HTTP request, outside the SmartThings platform, or from another SmartApp or Device Handler, we need to validate the input.

The preferred method of validation is to use a ``switch()`` statement on the input before doing anything with it:

.. code-block:: groovy

switch(mystring) {

case "cmd1":

object.cmd1()

break

case "cmd2":

object.cmd2()

break

case "cmd3":

object.cmd3()

break

default:

return "ERROR"

}

Do not hard-code SMS messages

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Notifications should never be sent to a hard-coded number.

They should always use a number provided by the user using the :ref:`contact input <contact\_book>` (even though Contact Book is not enabled, the contact input type is available and contains a fall-back mechanism for non-Contact Book users. Using this future-proofs your SmartApp).

----

Performance

-----------

Do not use busy loops

^^^^^^^^^^^^^^^^^^^^^

There is no good reason for the code to run busy loops.

Don't do things like this:

.. code-block:: groovy

def mywait(ms) {

def start = now()

while (now() < start + ms) {

// do nothing, just wait

}

}

The goal of the above code is to delay execution for a number of milliseconds.

This wastes resources and increases the likelihood that the 20 second execution limit will be exceeded.

Instead of trying to force a delay in execution, you should :ref:`schedule <smartapp-scheduling>` a future execution of your app.

Do not use ``synchronized()``

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Using ``synchronized`` incurs a performance overhead, and is highly unlikely to have any effect.

It should not be used.

When a SmartApp or Device Handler executes, it is executing on one of \*n\* available servers assigned for that Location, where \*n\* is variable depending on Location, current load, and other factors.

Concurrent executions of the SmartApp or Device Handler are not guaranteed, or even likely, to be executing on the same server.

Because of this, trying to force synchronous behavior by using ``synchronized`` would only work in the rare occurrence that a concurrent execution happens on the same server, yet it always incurs overhead.

----

LAN-specific

------------

Use the device-specific search

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Service managers for LAN-connected devices should :ref:`subscribe to the device search target <lan\_device\_discovery>` for device discovery.

Handle IP change

^^^^^^^^^^^^^^^^

Service managers for LAN-connected devices should :ref:`handle any IP change <lan\_device\_health>`.

This can happen when the router power cycles and loses its DHCP mappings.

----

.. \_review\_guidelines\_parent\_child:

Parent-child relationships

--------------------------

Use separate files

^^^^^^^^^^^^^^^^^^

When using a parent-child relationship, be it a parent SmartApp with child devices, or a parent SmartApp with child SmartApps, the parent and child should exist in separate files.

Putting the parent and child code in the same file leads to file size bloat, makes the code harder to understand, is error-prone, and difficult to debug.