FingerTips

A framework to model logical flows.

The implementation of *FingerTips* may seem unnecessary complex.

The intention of *FingerTips* is to move the complexity of the solution of its domain problem from its use to its implementation, hiding the complexity from its users, and to provide a simple, clear and unified interface of use.

This article details some of the implementation decisions taken during the design of *FingerTips*.

Table of Contents

1 – Flow state.

The building blocks of *FingerTips* are ***Flows***.

In its most simple form a ***Flow*** has an id and holds a single value.

The value can be any object.

The value can be read with

```
const value = flow.getValue()
```

and set with

```
const newValue = 1
flow.setValue(newValue)
```

2 – Nested flows.

A ***Flow*** can have 0 or more nested child flows.

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
    this.value({ id: 'child-1' })
    this.value({ id: 'child-2' })
    this.value({ id: 'child-3' })
})
```

A Flow can have any number of child flows but only one parent, making the main ***Flow*** to define a tree of ***Flows***.

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
    this.value({ id: 'child-1' }, function() {
          this.value({ id: 'child-1-1' })
          this.value({ id: 'child-1-2' })
     })

    this.value({ id: 'child-2' }, function() {
          this.value({ id: 'child-2-1' })
     })

    this.value({ id: 'child-3' })
})
```

3 – Child flows are unaware of its ascendants.

A ***Flow*** knows and can access its own children and descendants but it has no access or even is aware of its parent ***Flow***.

```
flow.getChildFlow({ id: ‘child-1’ })
flow.getChildFlow({ id: ‘child-2-1’ })
flow.getChildFlow({ id: ‘child-1.child-2-1’ })
```

This is a design decision to help achieve flows encapsulation, separation of concerns and security.

A ***Flow*** should not define its behavior based on the current state of its parent or siblings flows in the same way that a function should not define its behavior based on global variables.

It makes the logic more simple and secure and less error prone.

However it is still possible and simple to make a flow to access an ancestor through the use of a variable defined in the outer scope of the ***Flow*** definition:

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
    this.value({ id: 'child-1' }, function() {
          this.value({ id: 'child-1-1' })
          this.value({ id: 'child-1-2' })
     })

    this.value({ id: 'child-2' }, function() {
          thisFlow // the main flow object
          this.value({ id: 'child-2-1' })
     })

    this.value({ id: 'child-3' })
})
```

The decision of whether to access a flow ancestor or not is left to the developer.

4 – The update and synchronization of the *Flow* state is based on events.

When the state of a ***Flow*** changes that very same ***Flow*** receives a notification of the change event where it can update the state of its children or perform other actions based only on its own state.

This is a design decision to make the logic of the flows synchronization more simple and easy to track.

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {

     this.whenValueChanges( ({ newValue: n }) => {
          const child1 = this.getChildFlow({ id: ‘child-1’ })
          child1.setValue(n + 1)
```

```
    })

    this.value({ id: 'child-1' }, () => {

        this.whenValueChanges( ({ newValue: n }) => {
            console.log(n)
        })

    })

})
```

5 – Flows can export interfaces to other objects called FlowPoints.

One way of allowing other objects to interact with a ***Flow*** object is to give the other object a direct reference to the ***Flow*** object:

```
const flow = Flow.new()

const window = WindowComponent.new()
window.setFlow(flow)
```

and from there on the *window* can interact with the *flow* through its direct reference.

This approach has a major problem: it exposes all the protocol of the ***Flow*** to every object it has a reference to it. With this reference an object might even change the structure of a flow and modify its logic.

This may not be a concern if the object holding a reference to the flow is part of the same application than the referenced ***Flow*** but it would be if, for example, the application allows the use of plugins or libraries that require a reference to the flow.

The design decision to deal with this problem is not to expose the flow itself but a *Proxy* instead:

```
const flow = Flow.new()

const flowPoint = flow.asFlowPoint()


const window = WindowComponent.new()
window.setFlow( flowPoint )
```

A ***FlowPoint*** object is a *Proxy* to an actual ***Flow*** object with a limited set of functions to access the the ***Flow*** object.

By default the only protocol a ***FlowPoint*** has is to return the id of the ***Flow*** object

```
flowPoint.getId()
flowPoint.getIdPath()
```

and to ask for a child ***FlowPoint*** of itself

```
flowPoint.getFlowPoint({ id: ‘child-1’ }).
```

The rest of the methods in a ***FlowPoint*** to access the actual ***Flow*** are chosen dynamically by each ***Flow*** (the mechanism to grant methods to a ***FlowPoint*** will be improved in the future).

To ensure that the ***Flow*** is not accessible through any of its ***FlowPoints*** the ***FlowPoint*** does not include a reference to the ***Flow*** object, only the ***Flow*** methods granted to it.

So when building the ***FlowPoint*** object instead of doing

```
const flowPoint = FlowPoint.new()


flowPoint.flow = flow
```

```
flowPoint.doSomething = function(params) {
     flow.doSomething(params)
}
```

it does something equivalent to

```
const flowPoint = FlowPoint.new()

flowPoint.doSomething = flow.doSomething.bind(flow)
```

The actual implementation adds an extra wrapper but in no case a ***FlowPoint*** object holds no references to any ***Flow*** object.

6 – FlowPoints interface is used through regular method calls.

The commands exported by a ***Flow*** through its ***FlowPoints*** are called like regular javascript methods. This makes the ***FlowPoint*** interface fluent, simple to use and intuitive.

So instead of doing something like

```
Command.new({ id: ‘main.selectFile’ })
     .executeOn({ flowPoint: flowPoint })
```

it does

```
flowPoint.selectFile()
```

From the perspective of a user of a ***FlowPoint*** object the code does not change between having a reference to a ***FlowPoint*** of to a ***Flow*** object.

7 – The update of the Flow state to its FlowPoints is done through the Observer pattern.

When a ***Flow*** changes its state besides updating its child ***Flows*** state it needs to notify the change to its ***FlowPoints***.

This is done through an events mechanism:

```
const valueFlow = ValueFlow.new()


const flowPoint = valueFlow.asFlowPoint()


valueFlow.on(
     'value-changed',
     flowPoint.onFlowValueChanged.bind(flowPoint)
})
```

8 – The update of the Flow internal state and the update of the FlowPoints state is independent from each other.

When a ***Flow*** changes its state an event is received by the ***Flow*** itself. Also when a ***Flow*** changes its state it emits events to its ***FlowPoints***.

It is a design decision not to use the same events mechanism for both updates. The decision is motivated by several reasons.

The events mechanism may or may not guarantee the order of the notifications to its observers.

It can be deterministic or depend on the instant in which each ***FlowPoint*** is requested to the ***Flow***.

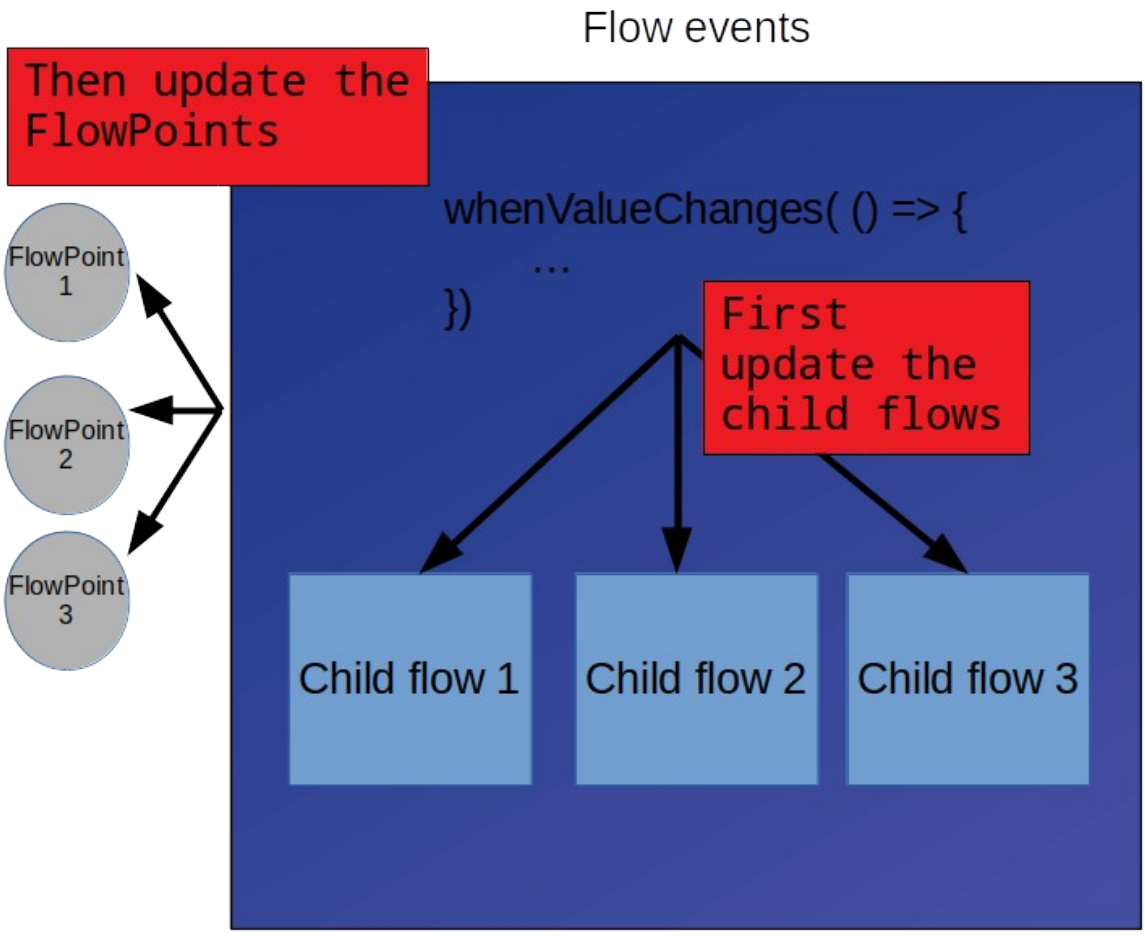
It can be synchronous or asynchronous. It can be delayed or immediate.

This is not an issue for the different ***FlowPoints*** of a ***Flow*** among each other but it is for the ***Flow*** itself.

If the order of the update notifications is not guaranteed it may happen that a ***FlowPoint*** is notified before the ***Flow*** executes its own update event. Since the ***FlowPoint*** may interact with the ***Flow*** it is mandatory for the ***Flow*** to update itself before updating its ***FlowPoints***.

Therefore calling the ***Flow*** update event before any ***FlowPoint*** update event is an invariant that is guaranteed for all ***Flows*** and its ***FlowPoints***.

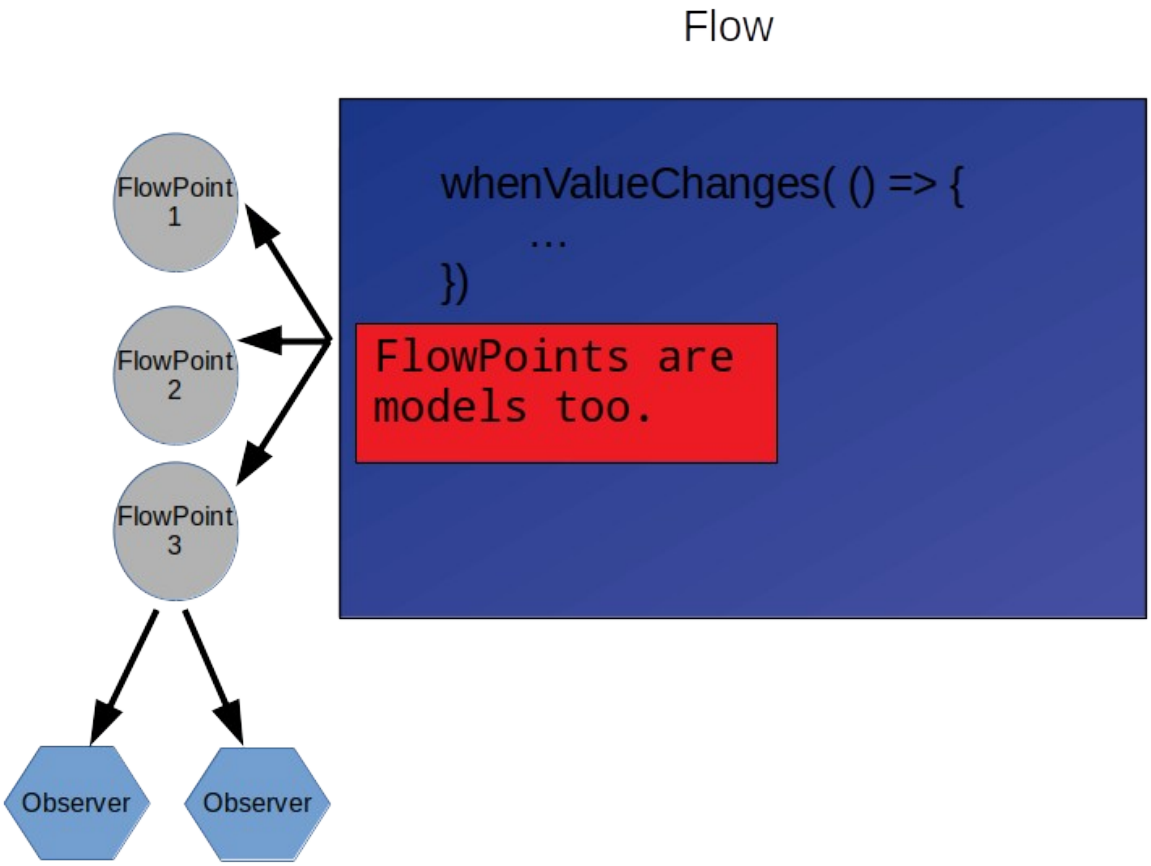
The mechanism to ensure this invariant is to use for all updates in the ***Flow*** two independent events mechanisms in a strict order: first one for the ***Flow*** object itself and then one for its ***FlowPoints***.



9 – Each FlowPoint is itself a Model.

A ***FlowPoint*** is updated when its ***Flow*** changes but it is a common case that other objects, such as a user interface or a subsystem implementing the *Chain of responsibility* pattern, would want to be notified of the changes as well.

To avoid the creation of yet another proxy on each ***FlowPoint*** to update the ***Flow*** state it is a design decision to make each ***FlowPoint*** a *Model* on itself, allowing *Observers* to subscribe to the changes the state of the ***Flow***.



10 – FlowPoints do not interact with its Flow calling its methods directly but through an implementation of the Command pattern.

One common problem in complex applications with user interfaces or dynamic plugins is to audit, control and filter the modifications and queries made to the application model. A subset of this problem is handling permissions and roles of users.

This problem is more simple to define, solve and trace if the application has a single entry point of execution for every possible command that the application performs on its model. In that single entry point the main application can audit, log, filter, validate, check permissions and have full control of the command before and after its execution.

It is a design decision to provide such an optional entry point of execution for all commands to the top most application level through the use of a light-weight *Command* pattern.

When a method is exported from a ***Flow*** to a ***FlowPoint*** what is exported is not the flow method itself but a wrapper on the method:

```
const flowPoint = FlowPoint.new()

// Gets all the commands exported by the flow
const flowCommands = flow.getDirectCommands()

flowCommands.forEach( (command) => {
     const commandMethodName = command.getId()

     flowPoint[commandMethodName] = function(...params) {
               return command.execute({ params: params })
          }
})
```

where the *Command.execute* method does not call the *flow* method directly but asks a ***CommandsController*** to execute itself instead

```
command.execute = function(...params) {
     const commandsController = this.getCommandsController()

     return commandsController.doExecuteCommand({
          command: this,
```

```
            params: params,
        }
}
```

So when a ***FlowPoint*** method is called

```
flowPoint.doSomething({ with: 1 })
```

quite a few things happen

- The ***flowPoint .doSomething({ with: 1 })*** method is called
- ***.doSomething({ with: 1 })*** method is bound to a ***Command*** object created by the ***Flow***
- The ***Command*** object gets its ***CommandsController***, defined by the ***Flow*** that created the ***Command***
- The ***Command*** object calls ***CommandsController.doExecuteCommand(…)*** with itself as a parameter
- The ***CommandsControllers*** may inspect the ***Command*** and, based on the current state of the ***Flow***, on the permissions, on the roles currently assigned to the user, etc, decide what to do with the command. By default it just executes it.

This mechanism also applies to the updates of the ***Flow*** itself, not only to the ***FlowPoints.***

```
flow.setValue( 1 )
```

goes through the main flow ***CommandsController*** as well.

This way the main ***Flow*** ensures that all updates to itself, at any level of nesting, goes through a single ***CommandsController*** defined by the top most level ***Flow*** and that all this complex mechanism is transparent to both the ***Flow*** definition and its ***FlowPoints***.

The only code that needs to be aware of this mechanism is the ***CommandsController*** itself, if one is defined.

11 – The CommandsController definition is optional.

It is a design decision not to force the creation of a ***CommandsController*** unless one is required by the main ***Flow***.

If no ***CommandsController*** is given the ***Commands*** are executed without any other processing.

12 – A child Flow can be factorized in an indepedent Flow object.

In most cases a developer would want not to have a monolithic ***Flow*** but to divide that logic in smaller, reusable and pluggable **Flows**, in the same way that she does with ***Classes*** and ***Functions***.

That is accomplished with the use of the ***flowBuilder.object()*** definintion:

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
     this.object({
          id: 'child-1',
          definedWith: CustomChildFlow.new(),
     })
})
```

instead of

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
    this.value({ id: 'child-1' }, function() {
          // ...
     })
})
```

Note that in the case of isolating a child ***Flow*** on its own class it no longer has access to the variables defined in the outer scope of the flow, like the variable '*thisFlow*', unless they are explicitly passed to the child ***Flow*** object as parameters during its instantiation.

This makes the use of child flows available in libraries and plugins more secure and less coupled with the parent ***Flow*** containing it.

13 – Each Flow can define its own CommandsController at any level, but if an ascendant Flow of it defines one the ascendant CommandsController is preferred and used.

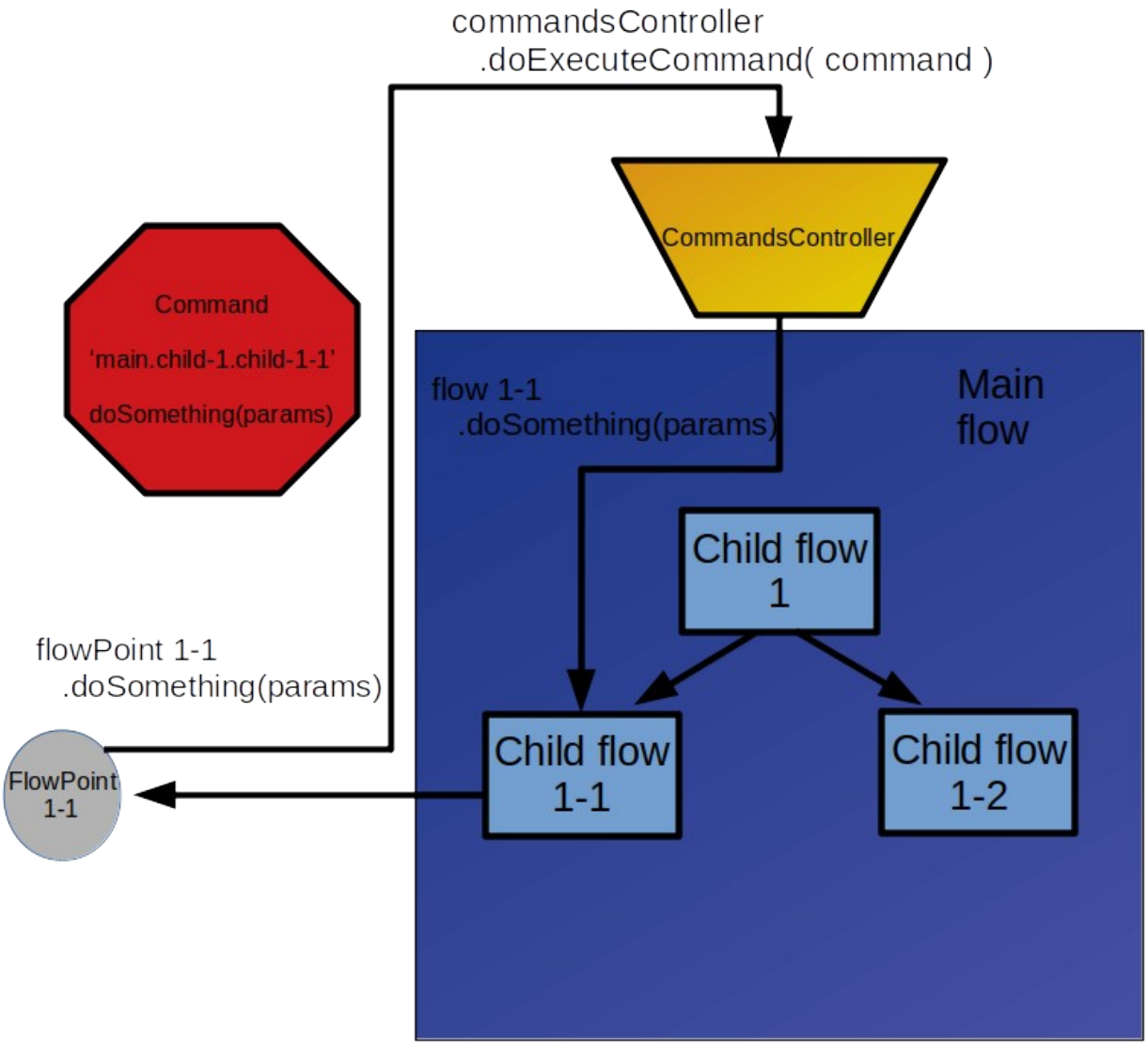
Since a ***Flow*** is unaware of its parent it can not rely on having or not a ***CommandsController*** given from an ascendant. In that case in may define and use its own ***CommandsController***.

However, when any ***Flow*** is added as a child of a parent ***Flow*** the ***CommandsController*** of the parent is set to the child, overriding any previous ***CommandsController*** the child might have.

```
addChildFlow({ id: childFlowId, flow: childFlow }) {
     childFlow.setCommandsController( this.commandsController )

     this.childFlows[childFlowId] = childFlow
}
```

This design decision ensures that when the top most ***Flow*** defines its own ***CommandsController*** it is used by all of the ***Flows*** in the tree, from the root to its leafs, no matter what each ***Flow*** would do in isolation, even if the ***Flows*** are added dynamically at any level in any instant.



14 – Each Flow in the tree has a unique id path.

Even though the id of a child ***Flow*** may be the same as another ***Flow*** id, each ***Flow*** is identified from the top most flow with a full path id.

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
     this.value({ id: 'child-1' })
})
```

```
childFlow.getId() == ‘child-1’
childFlow.getIdPath() == ‘main.child-1’
```

This makes it possible for the ascendant ***Flows*** and for the ***CommandsController*** to access any child ***Flow*** at any level in the tree if necessary.

15 – Commands are a particular case of a child Flow.

Commands are ***Flows*** too, except that they do not have child ***Flows*** and that they can be executed and exported as methods to a ***FlowPoint***.

In the future this may change and ***Commands*** might have children and define a tree of stateless flows.

16 – Child Commands are overridable by an ascendant Flow.

A parent ***Flow*** can override or replace a child ***Command*** at any level of nesting.

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
     this.object({
          id: 'library',
          definedWith: LibraryMainFlow.new(),
     })

     this.command({
          id: 'library.sendEmail',
          whenActioned: ({ email: email }) => {
               CustomMailer.new().sendEmail({ email: email })
          },
     })
```

```
})
```

17 – Child Flows can request to its ascendants to evaluate commands.

In some circumstances a child ***Flow*** exports a ***Command*** that is defined on its ascendants.

For example an application may have a logged user or not. Down in the ***Flow*** tree a component may want to show or not a widget depending on whether the application has a logged user or not.

The child ***Flow*** can be anywhere down in the tree, meaning that the application state should be reachable from the main ***Flow*** to the child ***Flow*** across all the intermediate flows.

Some possible approaches to make this state reachable to the child ***Flow*** are:

- The application state can be global to the entire tree.

```
class ChildFlow {
     getUser() {
          return globalApp.getUser()
     }
}
```

This approach is the less convenient since it exposes the global state to the entire tree and is has all the problems of using global variables.

- Each child ***Flow*** may know its parent and ask for the application state.

```
class ChildFlow {
     getUser() {
          return this.getParentFlow().getUser()
     }
}
```

This makes the child flows to be aware of its parent up to the root flow and have a direct reference to it, breaking the encapsulation of the child flow.

- The child ***Flow*** may request the application state using a loose coupled events mechanism.

```
class ChildFlow {
     getUser() {
          const eventData = { loggedUser: null }

          this.emit('loggedUser', eventData)

          return eventData.loggedUser
     }
}
```

This approach is complex to track since all parent flows must subscribe to the events of all of its children at any level.

- The application state can be pushed from the application down to the child ***Flow*** following the *Inversion of Control* pattern.

```
class ParentFlow {
     build() {
          ChildFlow.new({
               loggedUser: this.loggedUser
          })
     }
}

class ChildFlow {
     construct({ loggedUser: loggedUser }) {
```

```
            this.loggedUser = loggedUser
        }

        getUser() {
            return this.loggedUser
        }
}
```

This solution has many advantages over the previous ones: it preserves the child flows encapsulation, it exposes only the application state that each child requires, the child flow is unaware of its parent and can not reach its ancestors and it is traceable.

FingerTips accomplishes this requirement using a mechanism that allows a child ***Flow*** to delegate the execution of a ***Command*** through its ancestors path up to the main ***Flow*** without breaking the encapsulation with a solution that resembles the *Resumable Exception* pattern and that is somewhere in between the Events solution and the *Inversion of Control* solution.

Since all the flows in the tree have the same ***CommandsController*** and since the ***CommandsController*** knows the path from the child flow to the main flow, the child flow asks its ***CommandsController*** to bubble up a ***Command*** for execution without being aware of which ancestor flow will handle it.

The ***CommandsController*** walks the tree path up to the main flow asking each flow in the path if it handles that particular ***Command***.

When it finds the first ancestor ***Flow*** that handles the ***Command*** it asks to execute it and returns the result to the child ***Flow***.

Any flow can execute bubbled up commands but only the ones declared explicitly, otherwise all of its commands could be executed by any of this descendant in the tree, breaking the flows boundaries.

```
class ParentFlow {
    build() {
```

```
          this.acceptedBubbledUpCommands({
               commands: [
                    ‘getUser’,
               ]
          })
     }
}

class ChildFlow {
     getUser() {
          return this.bubbleUp({ command: ‘getUser’ })
     }
}
```

This implementation makes it simple to move a child flow from one place in the tree to a different one without modifying the intermediate flows, and allows the ***CommandsController*** to trace the bubbled up commands as well.

Its protocol allows to define a handler in case no parent handles the given command:

```
class ChildFlow {
     getUser() {
          return this.bubbleUp({
               command: ‘getUser’,
               ifUnhandled: () => { return this.getGuestUser() }
          })
     }
}
```

and allows a parent ***Flow*** to define a default handler for the unexpected bubbled up commands

```
class ParentFlow {
```

```
    build() {
        this.acceptedBubbledUpCommands({
            commands: [
                'someCommand',
            ],
            defaultHandler: () => {}
        })
    }
```

The last example would stop bubbling up the command to its ancestors to avoid a child flow to request for a ***Command*** beyond this `ParentFlow` ancestor.

18 – The *Dependency injection* mechanism can be achieved through the use of javascript variables defined in the outer scope of a Flow.

If a library or object provides a ***Flow*** with its logic the *Dependency injection* mechanism is achieved through the use of the *Command* pattern, javascript closures context binding and javascript scoped variables:

```
const singletonMailer = CustomMailer.new()

flowBuilder.main({ id: 'main' }, function(thisFlow) {
    this.object({
        id: 'library',
        definedWith: LibraryMainFlow.new(),
    })

    this.command({
        id: 'library.sendEmail',
        whenActioned: ({ email: email }) => {
```

```
                singletonMailer.sendEmail({ email: email })
          },
     })
})
```

19 – The design pattern *Inversion of control* is achieved through the Command pattern.

Using *FingerTips* ***Flows*** the pattern *Inversion of control* is achieved through the use of the ***Command*** pattern and allows different mechanisms of configuration and customizations.

- The top most ***Flow*** can define a custom ***CommandsController*** to use a custom ***Command*** instead of the default one provided by an object or library:

```
class AppCommandsController {
     doExecuteCommand({ command: command, params: params }) {
          if( command.getIdPath() === 'main.library.sendEmail' ) {
               CustomMailer.new().sendEmail(...params)
               return
          }

          return this.previousClassificationDo( () => {
               return this.doExecuteCommand({
                    command: command,
                    params: params
               })
          })
     }
}
```

- The top most ***Flow*** can override the nested ***Command*** provided by an object or library:

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
     this.object({
          id: 'library',
          definedWith: LibraryMainFlow.new(),
     })

     this.command({
          id: 'library.sendEmail',
          whenActioned: ({ email: email }) => {
               CustomMailer.new().sendEmail({ email: email })
          },
     })
})
```

- If the library or object implements the *Service Provider* pattern through the use of the *Command* pattern the top most ***Flow*** can accomplish the *Dependency Injection* mechanism through the *Command* pattern:

```
flowBuilder.main({ id: 'main' }, function(thisFlow) {
     this.object({
          id: 'library',
          definedWith: LibraryMainFlow.new(),
     })

     this.command({
          id: 'library.getMailer',
          whenActioned: ({ mailerConfig: mailerConfig }) => {
```

```
                return CustomMailer.new()
            },
        })
})
```

Note that if the overriden *'library.getMailer'* ***Command*** requires some extra context to instantiate the custom object the top most ***Flow*** can make it available to the closure with a scoped variable.