LABORATOIRE D'INFORMATIQUE DE GRENOBLE

AppsGate Client Technical Specifications

July, 22nd, 2014

Table of Contents

1 Introduction 4

2 Technology Dependencies 4

3 Client Global Architecture 5

3.1 Network Communication: Communicator 6

3.2 The Event dispatcher: Dispatcher 6

3.3 The AppsGate client Router: AppRouter 7

3.4 Modules 8

3.4.1 Module 8

3.4.2 Collection 8

3.4.3 Model 8

3.4.4 View 8

3.4.5 Communication 9

4 Communication details 9

4.1 Start sequence 9

4.2 Device notification 11

4.3 Places 11

4.4 Programs 11

5 The End-User Development Environment 19

5.1 The SPOK Grammar 19

5.2 EUDE Interpreter 20

5.2.1 Program representation (Abstract Syntax Tree) 20

5.2.2 Nodes 21

5.3 The SPOK Editor 29

5.3.1 Parser generation 29

5.3.2 User support and compilation 29

Table of Figures

Figure 1. Representation of the global structure of the AppsGate Smart Home application. 4
Figure 2. Representation of the global architecture of an AppsGate client. 6
Figure 3. Initialization of an AppsGate client. 10
Figure 4. Processing of a device-related notification (here, a new value for the device name). 12
Figure 5. Creation of a Place by the end-user. 13
Figure 6. Modification of a Place by the end-user 14
Figure 7. Deletion of a Place by the end-user. 15
Figure 8. Creation of a program by the end-user 16
Figure 9. Saving a program by the end-user. 17
Figure 10. Deletion of a Program by the end-user 18
Figure 11. UML class diagram of the SPOK interpreter. 20
Figure 12. Class diagram of a Node. 21
Figure 13. The components of the Program Module. 31
Figure 14. Refreshing a program editor view 31
Figure 15. Key press of the Smart Keyboard by the end-user 32

1 Introduction

This document describes the AppsGate client of the AppsGate Smart Home application. As shown in Figure 1, the AppsGate server is intended to run on a Set-Top-Box (STB) platform, and clients run on Windows, Mac OS X, and GNU/Linux as well as on mobile operating systems, typically iOS and Android. At the time of this writing, an AppsGate client can be accessed by most web browsers. User Interface (UI) components are primarily designed for use on desktops and laptops but tablets and smart phones are not excluded.

Figure 1. Representation of the global structure of the AppsGate Smart Home application.

The technologies used for implementing an AppsGate client are listed in Section 2. Sections 3 and 4 respectively present an overview of the global architecture of the client and the communication details. Section **Erreur ! Source du renvoi introuvable.** is dedicated to the implementation of the SPOK (Simple PrOgramming Kit) End-User Development Environment.

2 Technology Dependencies

An AppsGate client is implemented in HTML5 and JavaScript and uses the following libraries:

- **RequireJS** (http://requirejs.org). RequireJS is a JavaScript file and module loader that optimizes memory management by loading only the modules that are needed.
- **Backbone.js** (http://backbonejs.org). Backbone.js is used to structure the AppsGate client in a similar fashion to the MVC/PAC architectural design patterns so as to achieve separation of concerns between domain-dependent functions, UI issues, and dialogue control.
- **Underscore** (http://underscorejs.org). Underscore is used as a *utility-belt* library. For instance, Underscore enriches the standard JavaScript array with methods to sort, filter or search array elements. It also includes its own micro-templating solution. It comes as a dependency of Backbone.js.
- **jQuery** (http://jquery.com). jQuery is a JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for web development. It is used to manage DOM manipulations, animations and event bindings. It comes as a dependency of Backbone.js.
- **Bootstrap** (http://getbootstrap.com). Bootstrap is a web front-end framework that provides UI components. It also facilitates the implementation of responsive layouts as required for AppsGate since the targeted client platforms may have very different screen resolution and size.

- **PEG.js** (http://pegjs.majda.cz). PEG.js, formally Parser Generator for JavaScript, is a library used to generate the parser for the SPOK End-User Programming language.
- **Moment.js** (http://momentjs.com). Moment.js is a library to handle and format dates.
- **i18next** (http://i18next.com). i18next is a library to support internationalization. It allows an AppsGate client to be translated into several languages. At the moment of this writing, French and English are supported.

3 Client Global Architecture

As shown in Figure 2, an AppsGate client is structured into four groups of functions: the Communicator for exchanging information between the AppsGate server and the client, the Backbone core components, the AppRouter as a top level dialogue controller of the client, and a set of modules where one module corresponds to one class of user-centered domain-dependent concept. There are currently four such classes (therefore four modules): places, devices, services, and programs. In turn, each module is structured according to the Backbone architectural functional organisation, i.e. in terms of models, views and routers.

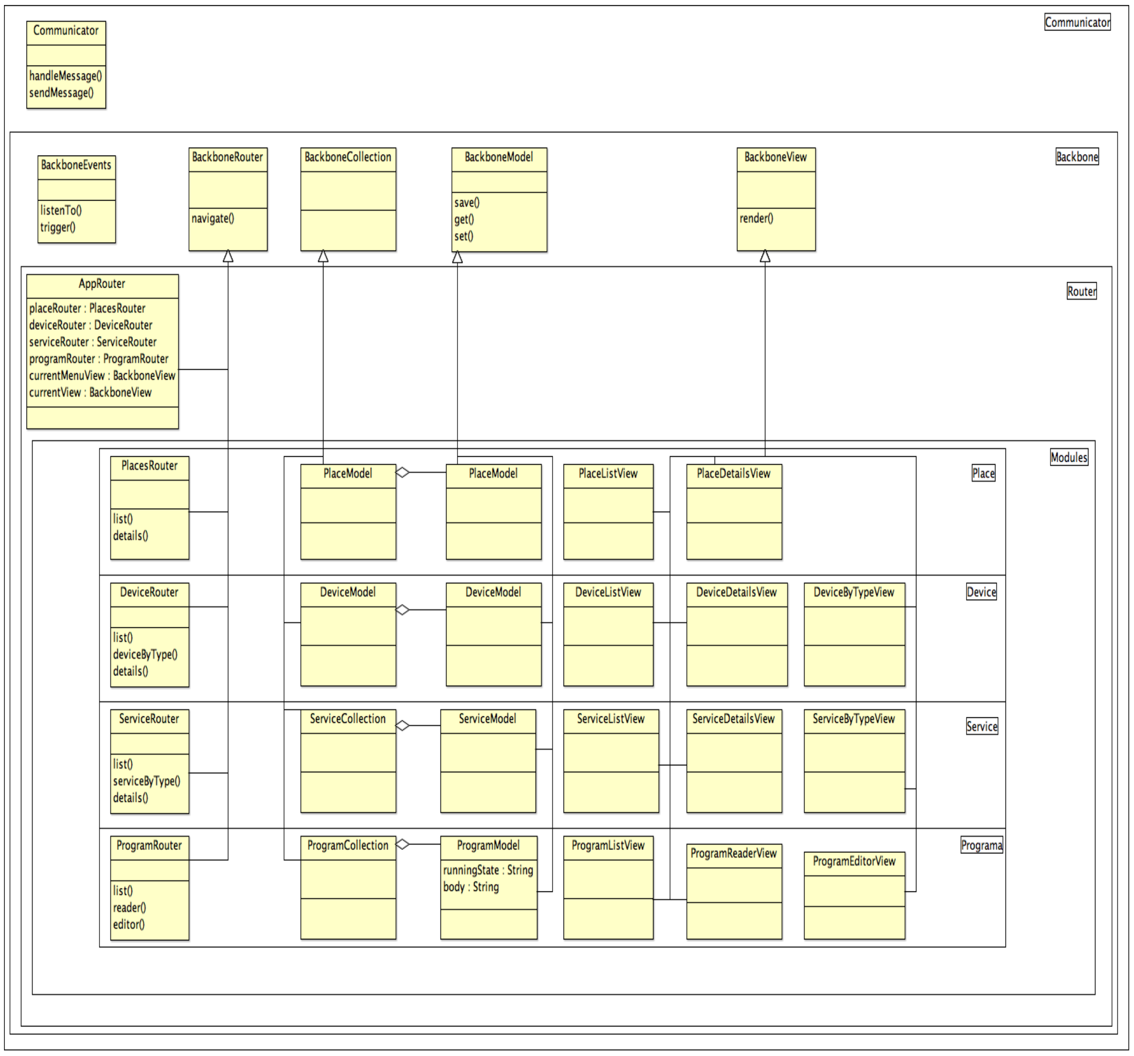


Figure 2. Representation of the global architecture of an AppsGate client.

3.1 Network Communication: Communicator

Communicator handles the communication between an AppsGate client and the AppsGate server. It is based on WebSockets and provides the AppsGate client with high-level methods to communicate with the server using the JSON serialization format. To send a message to the server, Communicator supplies the client with a method that formats data according to the AppsGate client-server protocol. When a message is received from the server, Communicator transforms the JSON message into the appropriate event by the way of Dispatcher, the event dispatcher.

Implementation: communicator.js

3.2 The Event dispatcher: Dispatcher

Dispatcher, which is based on *Backbone.Events*, provides a flexible mechanism to support communication between the components of an AppsGate client. It triggers

AppsGate client-specific events. Any component of the AppsGate client can subscribe to any event triggered by the event dispatcher. Typically, on receiving updates from the AppsGate server, Communicator uses Dispatcher to trigger events identified by the component's id slot specified in the update event. As Dispatcher allows components to subscribe to events whose id matches their own id, events are necessarily received by the appropriate client component.

In addition, Dispatcher is used in the Start sequence as detailed in Figure 3 to ensure that all needed data is received before the AppsGate client is fully operational.

Instantiation: app.js:28

3.3 The AppsGate client Router: AppRouter

A router is to Backbone what a Dialogue Controller is to Seeheim or what a Control is to PAC architectural conceptual models.

AppRouter plays the role of a top-level Dialogue Controller. It stores the state of the AppsGate client and manages event subscriptions expressed by the Views (more on Views, below). In addition, AppRouter handles navigation between the components through a set of routes that match the client-side pages by triggering the rendering of the correct page depending on the component the user sets the focus on. To achieve this, when changing the focus (i.e. when switching from the current page to a new one), AppRouter performs the following actions:

- Remove the View and all the events that were bound to that View
- Replace the View with the new one
- Renders the new View, attaches the events and makes it current

In addition, there is one specific implementation of the Backbone Router class for each of the user-centered domain-dependent module, to provide the appropriate navigation behaviour (or sub-dialogue control) within each module.

Instantiation: app.js:25
AppRouter implementation: router.js
PlaceRouter implementation: place.js
DeviceRouter implementation: device.js
ServiceRouter implementation: service.js
ProgramRouter implementation: program.js

3.4 Modules

3.4.1 Module

A Backbone module is a logical component that corresponds to a "user-centered domain-dependent" concept. At the time of this writing, four such modules are available: Place, Device, Service and Program.

A module is compliant with the Backbone MVC/PAC-like architectural design patterns. Adopting this organization has several advantages. First, it ensures low coupling between the functional core of the "user-centered domain-dependent" concepts and their concrete representations made available to users. Second, memory resource is optimized since modules are loaded dynamically according to the status of the AppsGate client.

3.4.2 Collection

In Backbone, a Collection is an ordered set of Models of the same type. For each Module *X*, a Collection is instantiated that contains the Model instances of the entities of class X detected/handled by the AppsGate server. For example, the Device module includes one Collection instance named *Devices* that contains *n instances of Device.Model* if *n* devices have been detected by the AppsGate server.

Places implementation: places.js
Devices implementation: devices.js
Services implementation: services.js
Programs implementation: programs.js

3.4.3 Model

In Backbone, Models are used to represent and manipulate data from the server that is of interest to end-users. The Model of an instance of class X contains a set of attributes that represent the current state of this instance and provides a set of functions to change (or get) its state as well as to synchronize it with the server side. Considering an instance of a smart plug, its Model contains data about the current power consumption, its name and the name of the place where the plug is installed, and whether it is switched on or off.

Place implementation: place.js
Device implementation: device.js
Service implementation: service.js
Program implementation: program.js

3.4.4 View

A View is a concrete representation of the state of a Model that is made available to the end-user for observation and manipulation. For example, in the case of the Device module, this representation is an HTML5 document generated by the *Device.View* using the *render* method. This method uses predefined HTML5 templates to generate this document. There are usually several Views per module to present data in different manners (e.g., as a list or as a detailed view for a device, or for programs, as a read-only or as an editable presentation).

Views implementations: Views

3.4.5 Communication

Communication within and across modules relies on Backbone events. When a Model is updated, the Model and the Collection it belongs to, fire a *change event*. By doing so, a state change can be triggered at different levels of granularity (i.e. at the collection level, and/or at the model level individually). A *Device.View* listens to events that occur from the Model it is interested in, as well as from UI elements of the generated HTML5 document, such as buttons and inputs. Modifications of the document are then translated by the View into modifications of the Model it corresponds to.

In order to synchronize the state of a Module with the state maintained by the AppsGate server, relevant Backbone methods are called by a Model when it is modified, such as *save*() or *destroy*() which in turn fires a Backbone *sync event*. The Model then synchronizes with the Appsgate server to update the objects it corresponds to on the server side. It also fires a Backbone.js *change event* that can be caught by the corresponding Views and/or Collection. Details are presented in the next section.

4 Communication details

This section details event routing within an AppsGate client as well as with the AppsGate server.

4.1 Start sequence

Figure 3 shows the sequence of operations that are performed when launching an AppsGate client. When started, the AppsGate client displays a splash screen to prevent the user from using the AppsGate client before it is ready. Then, it sends three requests to the AppsGate server to retrieve the devices, places and programs that currently exist. At the time of this writing, devices and services are returned as the result of one single “get devices” request to the server. It is up to the client to sort services and devices to create the appropriate service and device instances.

The AppsGate server returns three lists. (NB: responses are asynchronous and can be received in any order). On receiving a list of type X, the corresponding Models are instantiated and grouped into the Collection that corresponds to type X (either Places, Devices, Services, or Programs). Once places, devices, services, and programs are initialized, the AppsGate client is ready for use, and the splash screen is removed.

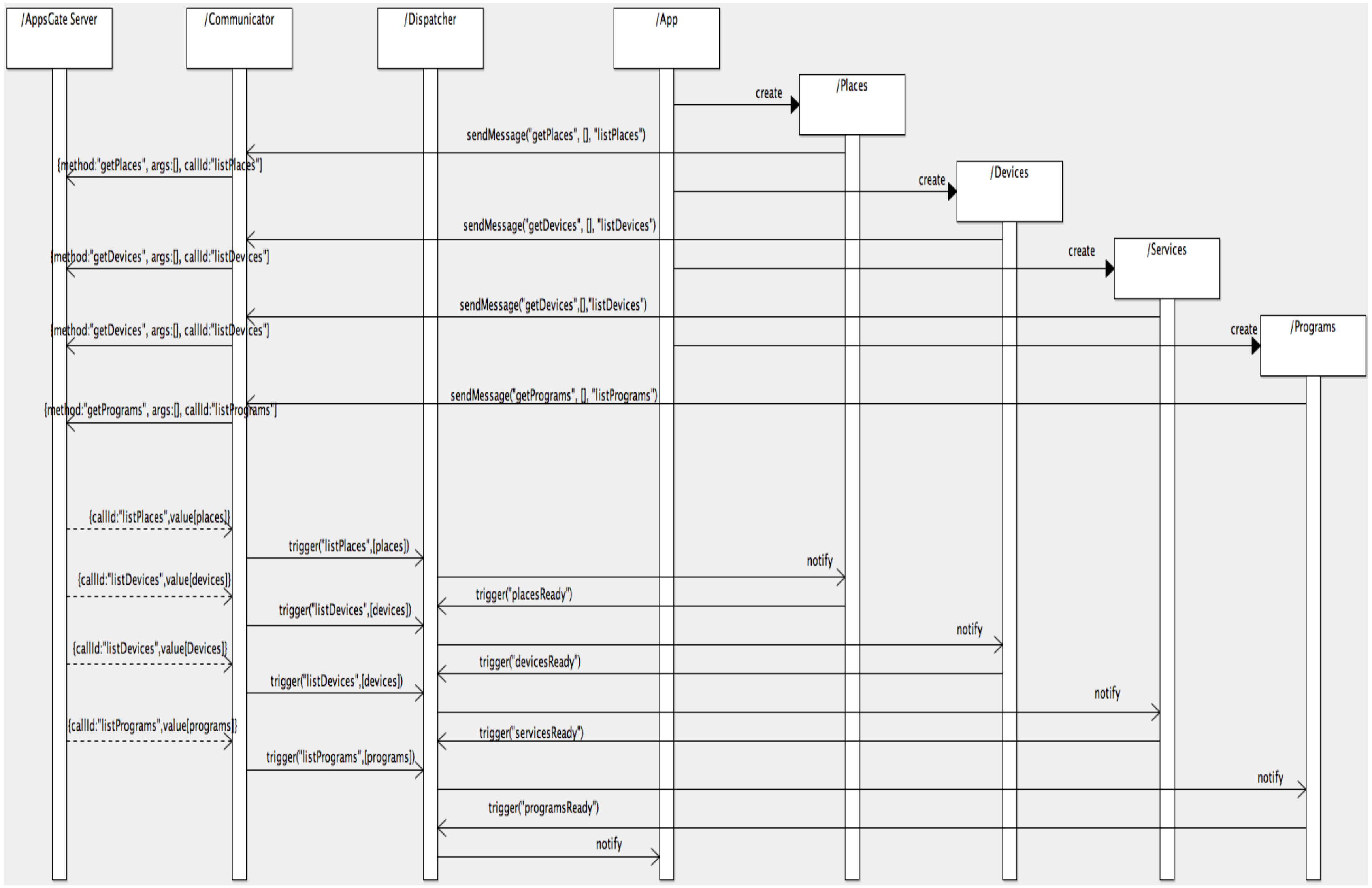

/AppsGate Server
/Communicator
/Dispatcher
/App
create
/Places
sendMessage("getPlaces", [], "listPlaces")
{method:"getPlaces", args:[], callId:"listPlaces"]
create
/Devices
sendMessage("getDevices", [], "listDevices")
{method:"getDevices", args:[], callId:"listDevices"]
create
/Services
sendMessage("getDevices",[],"listDevices")
{method:"getDevices", args:[], callId:"listDevices"]
create
/Programs
sendMessage("getPrograms", [], "listPrograms")
{method:"getPrograms", args:[], callId:"listPrograms"]
{callId:"listPlaces",value[places]}
trigger("listPlaces",[places])
notify
{callId:"listDevices",value[devices]}
trigger("placesReady")
trigger("listDevices",[devices])
notify
{callId:"listDevices",value[Devices]}
trigger("devicesReady")
trigger("listDevices",[devices])
notify
{callId:"listPrograms",value[programs]}
trigger("servicesReady")
trigger("listPrograms",[programs])
notify
trigger("programsReady")
notify

Figure 3. Initialization of an AppsGate client.

4.2 Device notification

The sequence diagram in Figure 4 shows how an AppsGate client processes a device-related notification from the AppsGate server. In this example, the AppsGate server notifies the AppsGate client that the name of the device *id* has the value *value*.

As explained above, the notification is expressed as a JSON message that is received by Communicator via a WebSocket. Communicator translates the message into a Backbone event that is then received by Dispatcher. The *id* element of the event allows the Dispatcher to identify the Model concerned by the notification. The Model updates its state accordingly (here, sets the new name), uses the Backbone *change* event to express its state change. Its Collection as well as all the Views that represent the Model, are notified of an update through the Backbone *change event*.

4.3 Places

The sequence diagrams of Figure 5, Figure 6, Figure 7,respectively illustrates the communication between the AppsGate client components when creating, updating and removing a place. The same interaction sequence applies to devices and services.

4.4 Programs

The sequence diagrams of Figure 8, Figure 9, Figure 10, respectively illustrates the communication between the AppsGate client components when creating, updating and removing a program.

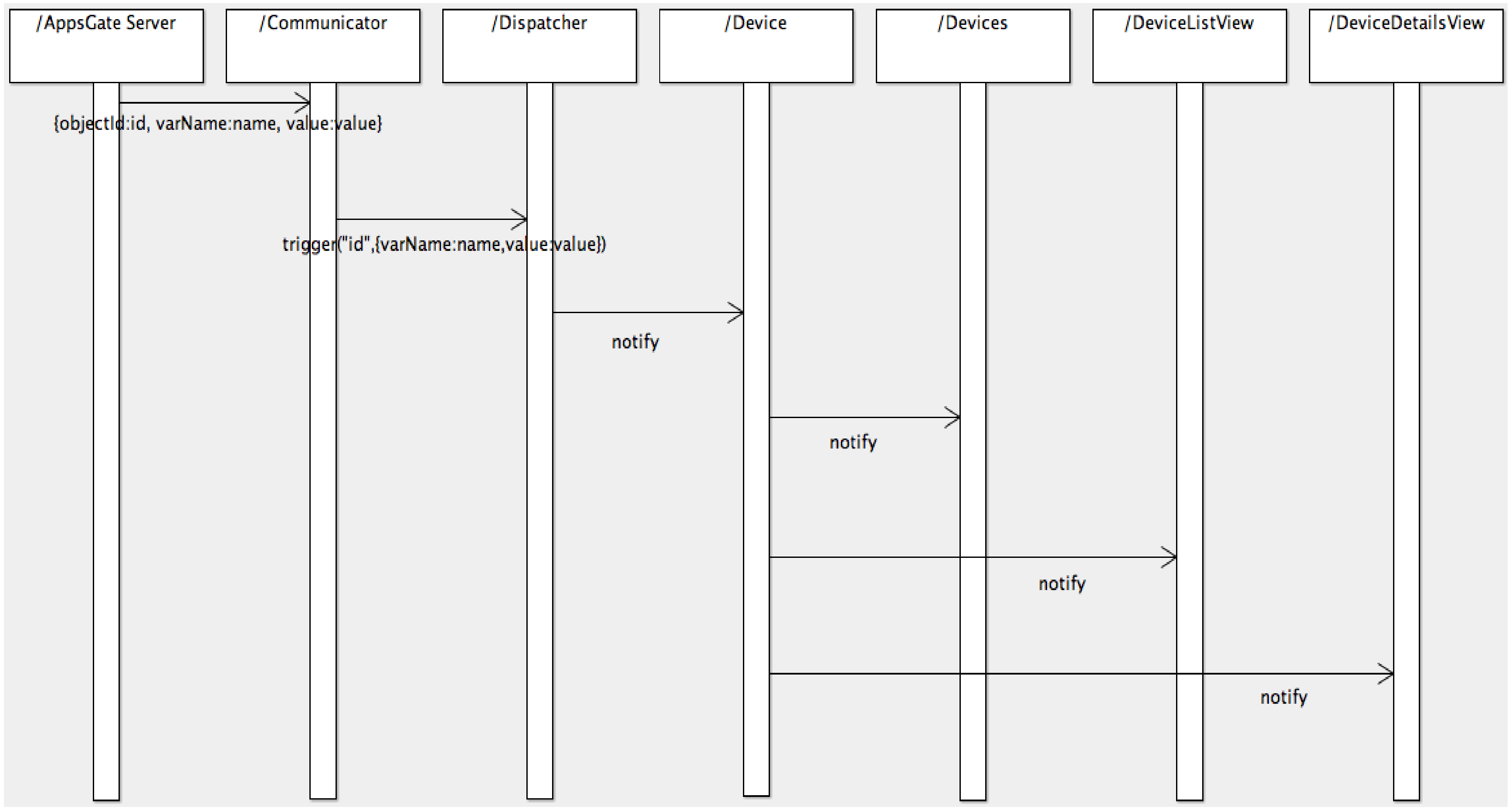


Figure 4. Processing of a device-related notification (here, a new value for the device name).

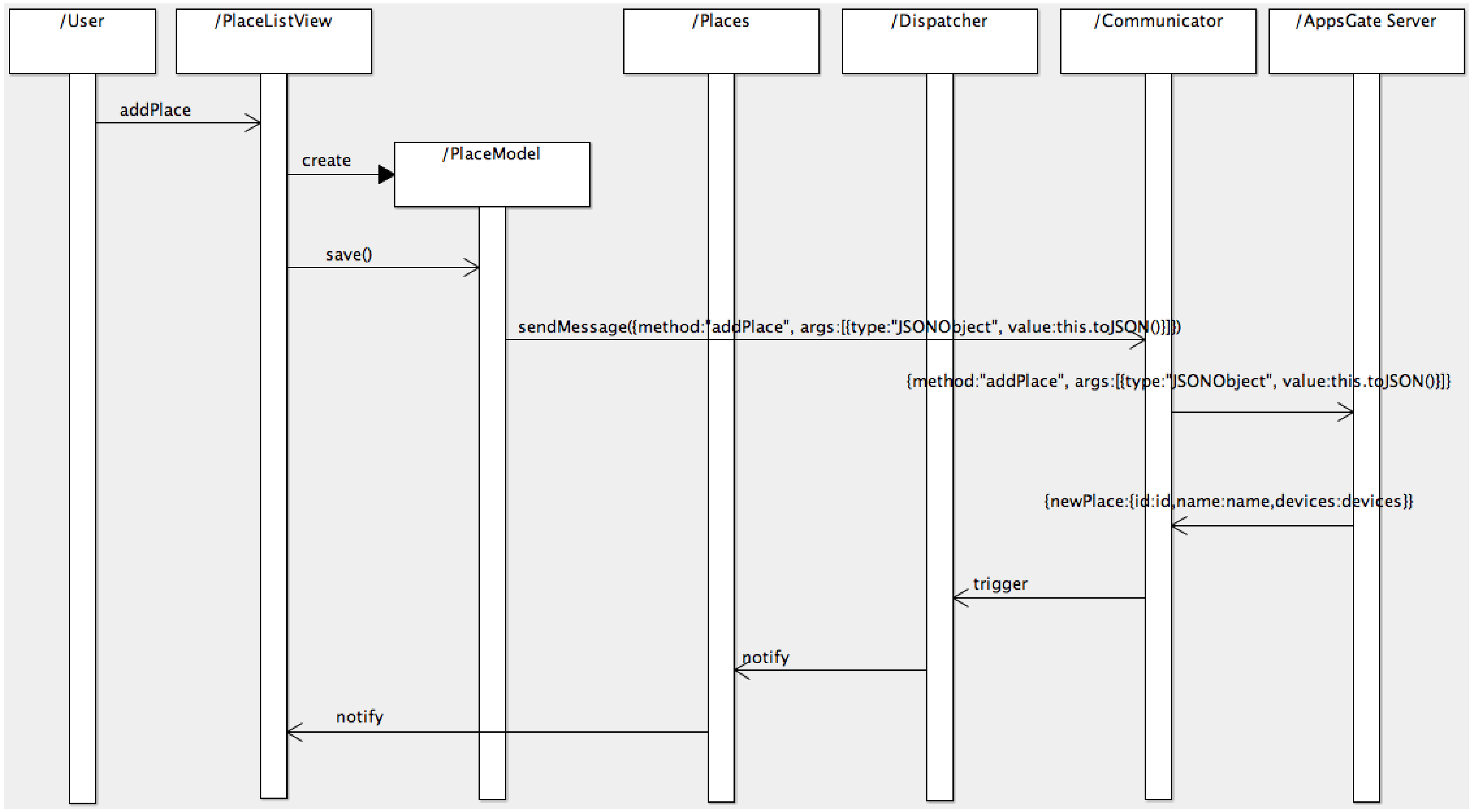

/User
/PlaceListView
/Places
/Dispatcher
/Communicator
/AppsGate Server
addPlace
create
/PlaceModel
save()
sendMessage({method:"addPlace", args:[{type:"JSONObject", value:this.toJSON()}]})
{method:"addPlace", args:[{type:"JSONObject", value:this.toJSON()}]}
{newPlace:{id:id,name:name,devices:devices}}
trigger
notify
notify

Figure 5. Creation of a Place by the end-user.

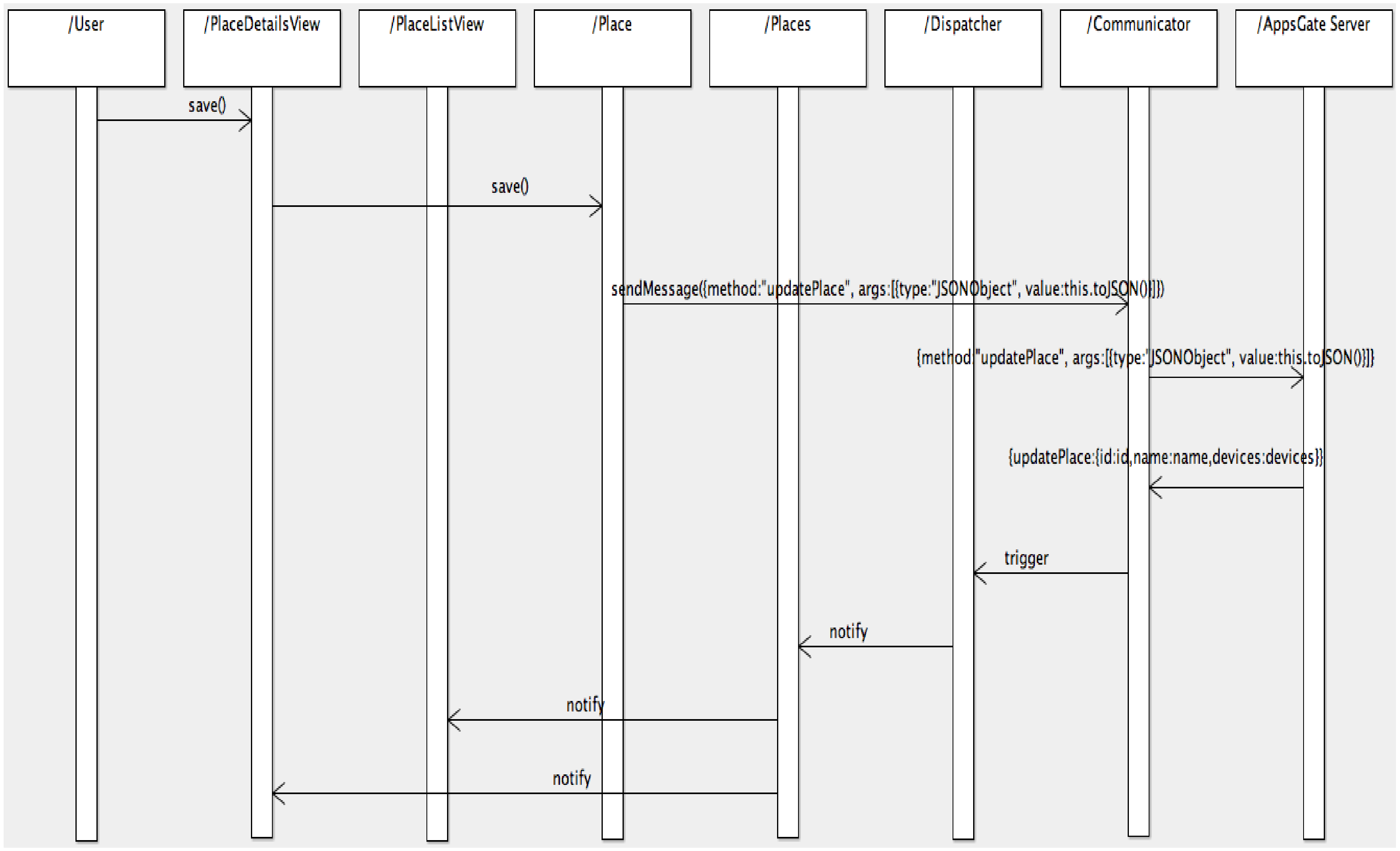

/User
/PlaceDetailsView
/PlaceListView
/Place
/Places
/Dispatcher
/Communicator
/AppsGate Server
save()
save()
sendMessage({method:"updatePlace", args:[{type:"JSONObject", value:this.toJSON()}]})
{method:"updatePlace", args:[{type:"JSONObject", value:this.toJSON()}]}
{updatePlace:{id:id,name:name,devices:devices}}
trigger
notify
notify
notify

Figure 6. Modification of a Place by the end-user

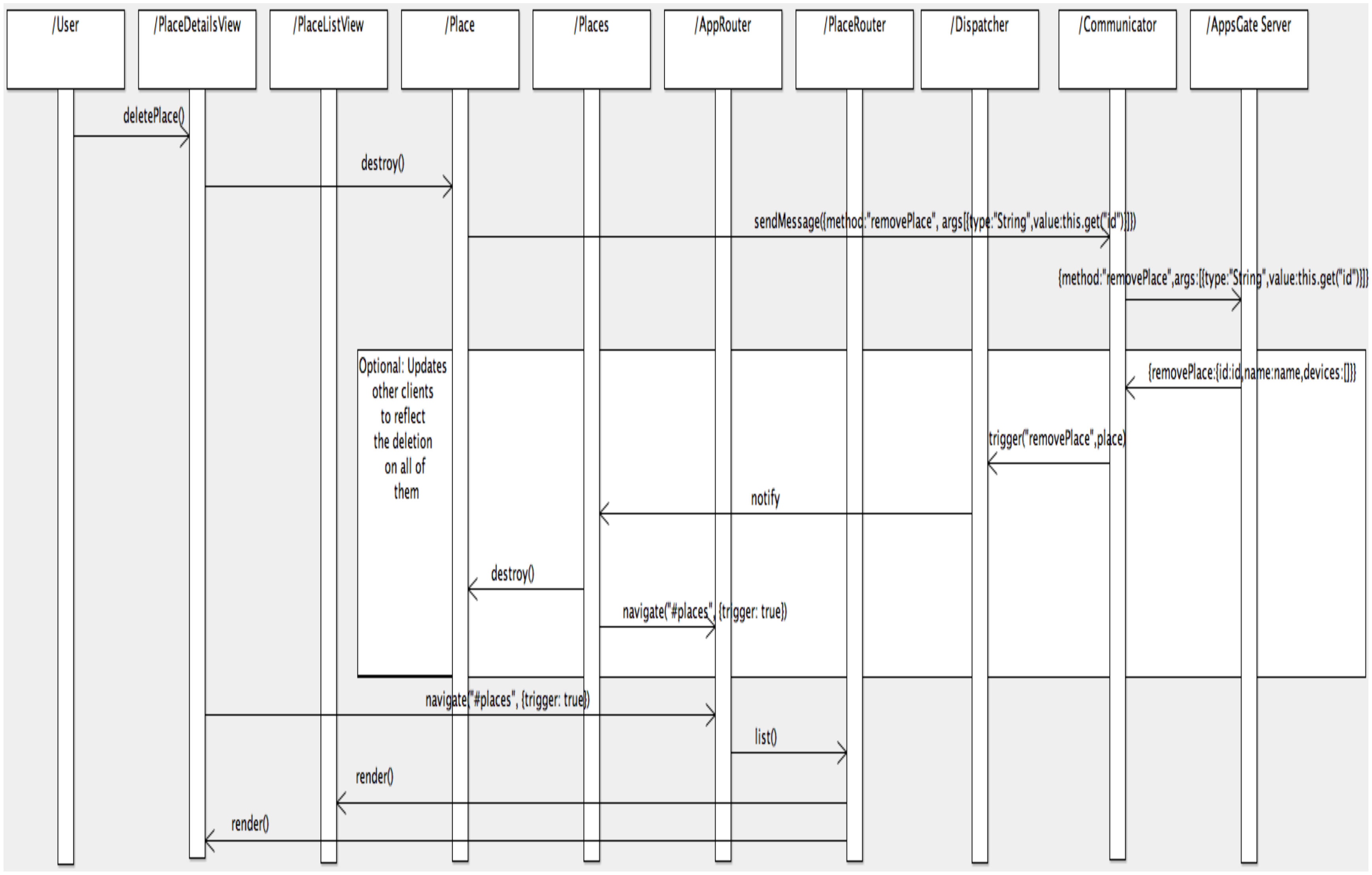

/User
/PlaceDetailsView
/PlaceListView
/Place
/Places
/AppRouter
/PlaceRouter
/Dispatcher
/Communicator
/AppsGate Server
deletePlace()
destroy()
sendMessage({method:"removePlace", args[{type:"String",value:this.get("id")}]})
{method:"removePlace",args:[{type:"String",value:this.get("id")}]}
Optional: Updates other clients to reflect the deletion on all of them
{removePlace:{id:id,name:name,devices:[]}}
trigger("removePlace",place)
notify
destroy()
navigate("#places", {trigger: true})
navigate("#places", {trigger: true})
list()
render()
render()

Figure 7. Deletion of a Place by the end-user.

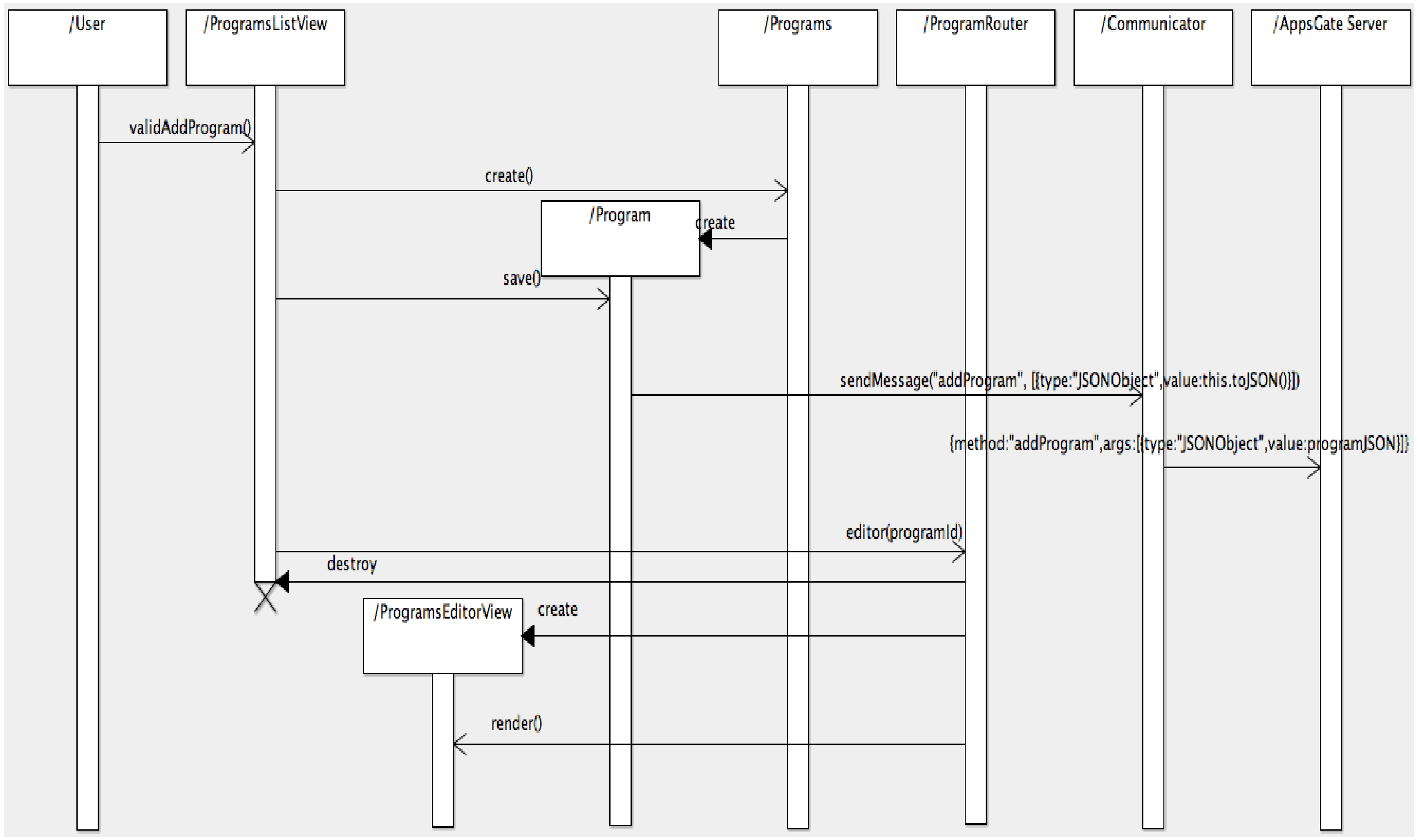


Figure 8. Creation of a program by the end-user

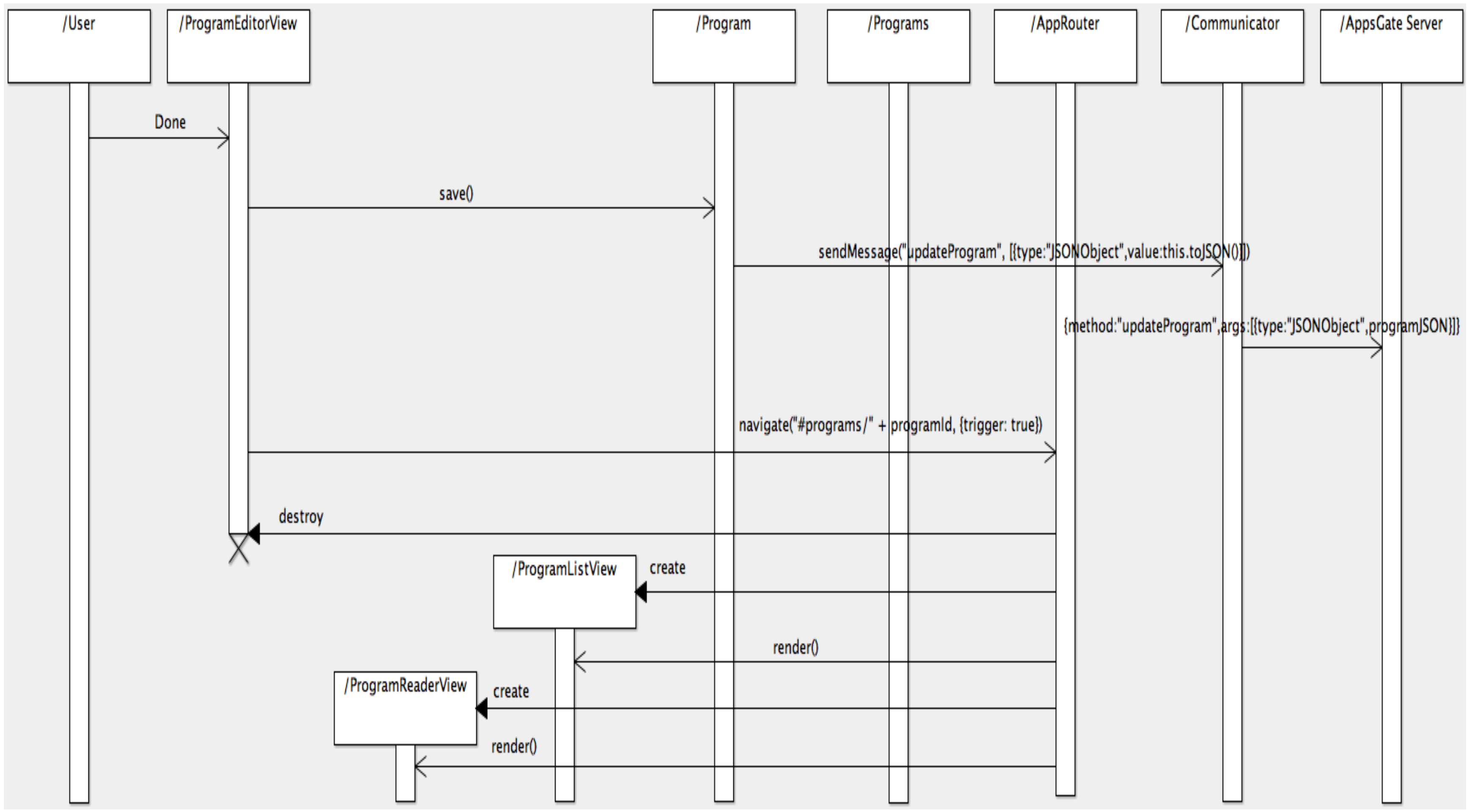

/User
/ProgramEditorView
/Program
/Programs
/AppRouter
/Communicator
/AppsGate Server
Done
save()
sendMessage("updateProgram", [{type:"JSONObject",value:this.toJSON()}])
{method:"updateProgram",args:[{type:"JSONObject",programJSON}]}
navigate("#programs/" + programId, {trigger: true})
destroy
/ProgramListView
create
render()
/ProgramReaderView
create
render()

Figure 9. Saving a program by the end-user.

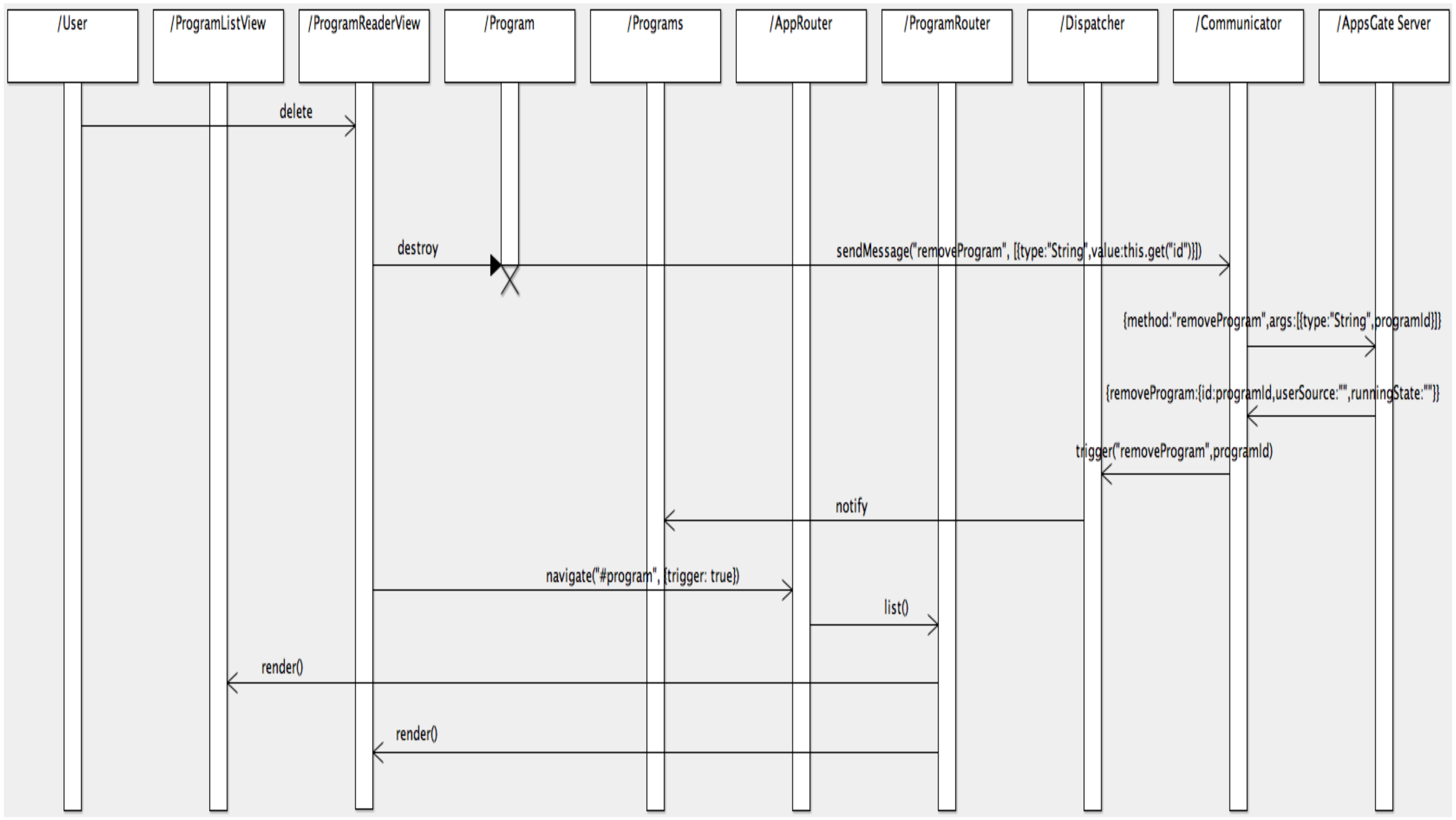

/User
/ProgramListView
/ProgramReaderView
/Program
/Programs
/AppRouter
/ProgramRouter
/Dispatcher
/Communicator
/AppsGate Server
delete
destroy
sendMessage("removeProgram", [{type:"String",value:this.get("id")}])
{method:"removeProgram",args:[{type:"String",programId}]}
{removeProgram:{id:programId,userSource:"",runningState:""}}
trigger("removeProgram",programId)
notify
navigate("#program", {trigger: true})
list()
render()
render()

Figure 10. Deletion of a Program by the end-user

5 The End-User Development Environment

SPOK (Simple PrOgramming Kit) is the AppsGate End-User Development Environment (EUDE). Ideally, an EUDE includes the functions that are necessary and sufficient for developing, testing, debugging, reusing, sharing, and maintaining programs. The particularity of a EUDE is that end-users are not computer science professionals. Their goal is to develop programs to simplify their daily activities with the help of the technology, not to learn programming and software engineering techniques.

In its current implementation, SPOK is comprised of an editor for building and modifying end-user programs, and an interpreter for executing end-user programs. The interpreter runs on the AppsGate server whereas the editor runs on a client end-user device. These are presented next. We start with the grammar as it is shared by the editor and the interpreter.

5.1 The SPOK Grammar

The language used by end-users to express their programs is a pseudo-natural language using a mix of rule-based and imperative programming. The left hand side of a rule is composed of events and conditions, and the right hand side specifies the actions to be taken when the left hand-side is true or becomes true. A program may include several rules that can be executed either in parallel or sequentially.

The BNF of the grammar currently implemented is the following:

```
SET = "setOfRules" _ "[" _ INSTS _ "]"
SEQ = "seqRules" _ "[" _ INSTS _ "]"
INSTS  = _ (INST _)+ / "(" INSTS ")"
INST  = IF / WHEN / WHILE / ACTION / EMPTY / WAIT
IF  = "if(" BOOLEAN ")" _ "(" _ BLOC _ ")" _ "(" _ BLOC _ ")"
    / "if" _ "(" BOOLEAN ")" _ "(" _ BLOC _ ")"
WHEN  = "when" _ "(" _ EVT _ ")" _ "(" _ BLOC _")"
WHILE  = "while" _ "(" _ STATE _ ")" _ "(" _ KEEP _ ")" _ "(" _ BLOC _ ")"
BLOC_KEEP = "seqRules" _ "[" _ (KEEP _)+ "]"
EVENTS  = "eventsSequence" _ "[" _ EVENTLIST _ "]"
        / "eventsAnd" _ "[" _ EVENTLIST _ "]"
        / "eventsOr" _ "[" _ EVENTLIST _ "]"
        / EVT
EVENTLIST = (EVENTS _)+
EVT  = "event" _ "(" _ SOURCE _ ")"
WAIT = "wait" _ "(" _ NUMERIC_VALUE _ ")"
BOOLEAN = "booleanExpression" _"(" _ BOOLEAN _ ")" _ "(" _ BOOLEAN _ ")"
         / "comparator" _"(" _ NUMERIC_VALUE _ ")" _ "(" _ NUMERIC_VALUE _ ")"
         / "booleanExpression" _"(" _ BOOLEAN _ ")"
         / STATE
NUMERIC_VALUE = "number"
              / "boolean"
             / PROPERTY
ACTION = "action" _ "(" _ TARGET _ ")"
PROPERTY = "property" _ "(" _ SOURCE _ ")"
KEEP  = "keepState" _ "(" _ MAINTANABLE_STATE _ ")"
STATE = "state" _ "(" _ SOURCE _")"
```

```
MAINTANABLE_STATE = "maintanableState" "(" _ TARGET _ ")"
TARGET = "variable" / "select" / "device" / "service" / "programCall" XXProgram”
SOURCE = "device" / "service" / "programCall"
VALUE = ID "string" / ID "number" / ID "boolean"
```

5.2 EUDE Interpreter

The SPOK interpreter is an ApAM component that runs on the AppsGate server. It manages a list of end-user programs and exposes the methods to add, remove, update, start and stop end-user programs.

The interpreter manages end-user programs using a hash map where the key is the end-user program *id* and the value is the abstract syntax tree (AST) of the end-user program. ASTs are stored in the JSON text format as the result of the compilation performed by the editor on the client side. The interpreter supposes that ASTs are semantically correct (no semantic error checking is performed). Figure 11 shows the class diagram of the SPOK interpreter.

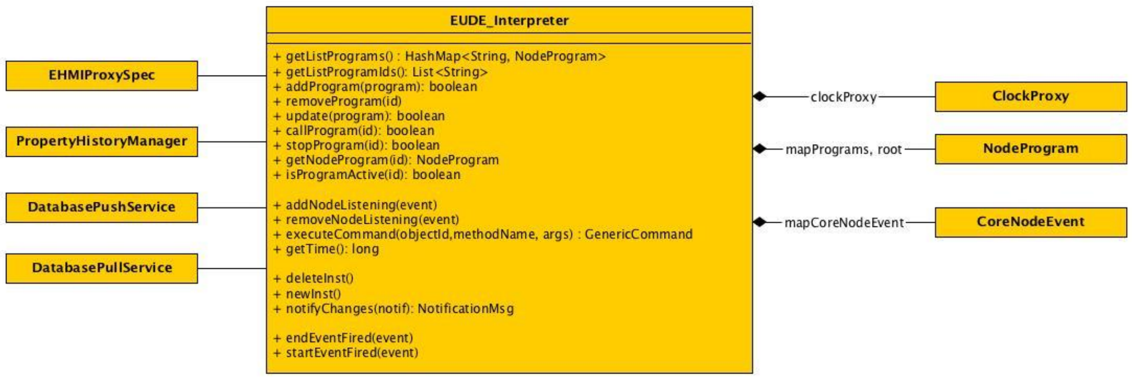

EUDE_Interpreter
+ getListPrograms() : HashMap<String, NodeProgram>
+ getListProgramIds(): List<String>
+ addProgram(program): boolean
+ removeProgram(id)
+ update(program): boolean
+ callProgram(id): boolean
+ stopProgram(id): boolean
+ getNodeProgram(id): NodeProgram
+ isProgramActive(id): boolean
+ addNodeListening(event)
+ removeNodeListening(event)
+ executeCommand(objectId,methodName, args) : GenericCommand
+ getTime(): long
+ deleteInst()
+ newInst()
+ notifyChanges(notif): NotificationMsg
+ endEventFired(event)
+ startEventFired(event)
EHMIProxySpec
PropertyHistoryManager
DatabasePushService
DatabasePullService
clockProxy
ClockProxy
mapPrograms, root
NodeProgram
mapCoreNodeEvent
CoreNodeEvent

Figure 11. UML class diagram of the SPOK interpreter.

5.2.1 Program representation (Abstract Syntax Tree)

The root node of an AST is a node whose type is NodeProgram. The children of a NodeProgram are of type NodeSetOfRules. The types of the children of a NodeSetOfRules include NodeIf, NodeWhen, NodeWhile and NodeAction. The children of a NodeSetOfRules node are processed in parallel whereas the children of NodeIf, NodeWhen, and NodeWhile are processed sequentially.

More precisely, a node can be one of the following types:
- NodeProgram: denotes the root node of an AST.
- NodeSetOfRules: a set of rules to apply.
- NodeWhen: processes the NodeSeqRules child on the occurrence of the specific events described by the NodeEvents child.
- NodeAction: is a call to a device (switch a lamp on or off), or a call to another program.

- NodeIf: evaluates the NodeBooleanExpression, then, applies one or the other NodeSeqRules.
- NodeWhile: evaluates a state, and maintains another state.

5.2.2 Nodes

This section details the nodes used to represent SPOK Abstract Syntax Trees, how they are implemented, and their JSON representations.
All Nodes inherits from the Node class. This class implements some interface and implements some methods that are useful for the nodes.
As shown in Figure 12, each Node has a parent Node, this way you can see the structure of the AST. Each Node also implements methods to manage start events and end events, which will be generated each time the node is called and once it is ended.
Node also implements interface Callable, but as Node is an abstract class, this method has to be implemented by each Node.

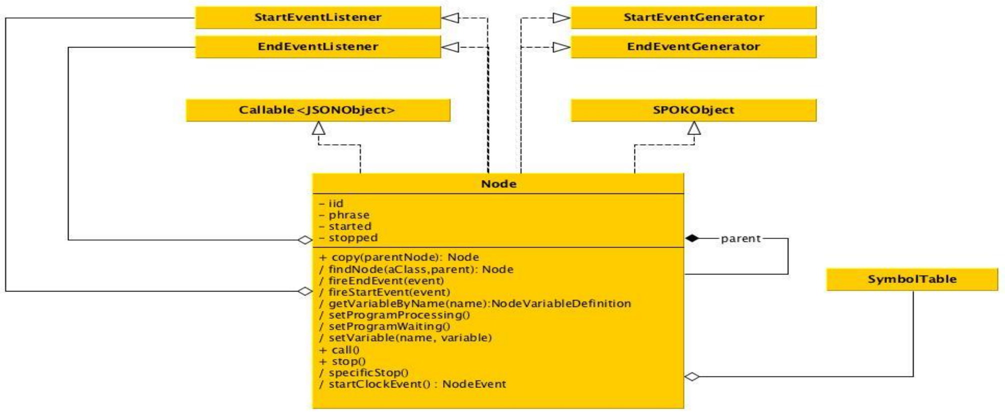


Figure 12. Class diagram of a Node.

5.2.2.1NodeProgram

JSON format of a *NodeProgram*:

```
{
    “type”: “program”,
    "id": "imb1",
    "name": "empty test",
    "parameters": [],
    "header": {
        "author": "Bob"
    },
    "definitions": [],
    "body": {
    }
}
```

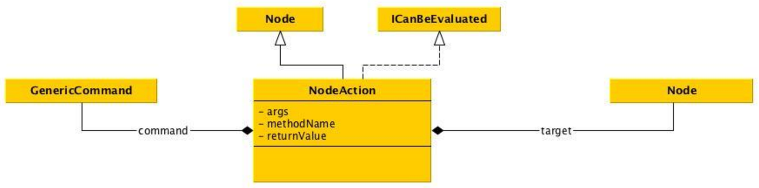
Where:

- *type* is program
- *id* is the program ID (unique)

- *name* is the name of the program, to show in the editor
- *header* is a JSON object that contains values that can help the understanding (ie: *author* is the user's name who has created the program).
- *body* is the AST of the program. It is a JSON object.
- *definitions* is a JSONArray that contains the variable definitions
- *parameters* is a JSONArray that contains the name of the parameters that will be used in the program

Implementation: NodeProgram.java

5.2.2.2 NodeAction



JSON format of a *NodeAction*:

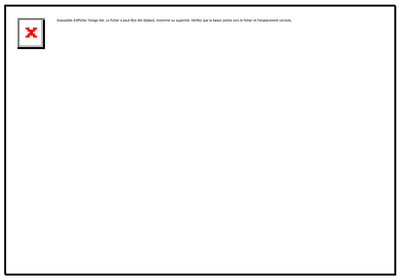
```
{
      "type": "action",
      "target": {
            "type": "device",
            "value": "test"
      },
      "methodName": "On"
}
```

Where:

- *type* is action
- *target*
  - *type* defines the type of the object on which the method call will be performed. Possible values are *program* and *device*.
  - *value* is the ID of the targeted object. It can be a device ID or a program ID.
- *methodName* is the name of the method to be called on the object.

Implementation: NodeAction.java

5.2.2.3 NodeSeqRules


Impossible d'afficher l'image liée. Le fichier a peut-être été déplacé, renommé ou supprimé. Vérifiez que la liaison pointe vers le fichier et l'emplacement corrects.

This is the basic node. All the node happens in sequence. It is represented like a list of nodes.

JSON format of a *NodeSeqRules*

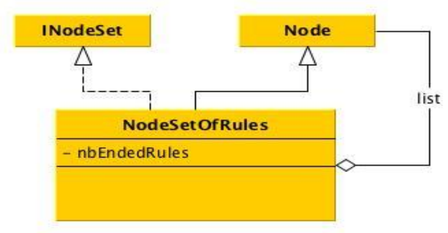
```
{
    "type": "seqRules",
    "rules": [
        {}
    ]
}
```

Where:

- *type* is seqRules
- *rules* is a JSONArray that contains Node

Implementation: NodeSeqRules.java

5.2.2.4 NodeSetOfRules



This one is different from the NodeSeqRules since it will launch every action without waiting the end of the previous node in the sequence.

JSON format of a *NodeSeqRules*

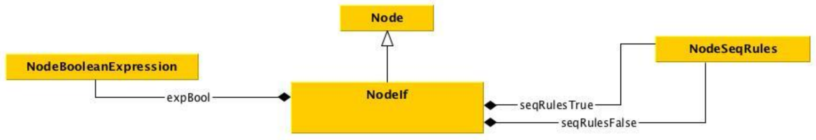
```
{
    "type": "setOfRules",
    "rules": [
        {}
    ]
}
```

Where:

- *type* is setOfRules
- *rules* is a JSONArray that contains Node

Implementation: NodeSetOfRules.java

5.2.2.5NodeIf



JSON format of a *NodeIf*:

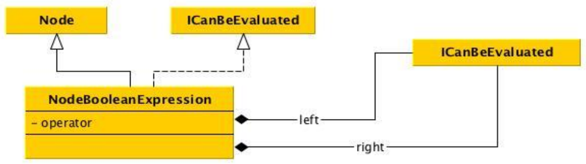
```
{
    "type": "if",
    "expBool": {
        "type": "booleanExpression",
        "operator": "==",
        "leftOperand": {
        },
        "rightOperand": {
        }
    },
    "seqRulesTrue": {
    },
    "seqRulesFalse": {
    }
}
```

Where:

- *type* is if.
- *expBool* is the node to be evaluated. It can either be a NodeComparator or a NodeBooleanExpression
- *seqRulesTrue* is the sequence of rules to evaluate if the boolean expression has returned *true*. It is a *NodeSeqRules* in its JSON representation.
- *seqRulesFalse* is the sequence of rules to evaluate if the boolean expression has returned *false*. It is a *NodeSeqRules* in its JSON representation.

Implementation: NodeIf.java

5.2.2.6NodeBooleanExpression



This node is evaluated by a NodeIf.
JSON format of a *NodeBooleanExpression*:

```
{
      "type": "booleanExpression",
	"operator": "&&",
	"leftOperand": {},
	"rightOperand": {}
}
```

Where:

- *type* is booleanExpression.
- *operator* is && or || or !
- *leftOperand* is a Node which can be evaluated
- *rightOperand* is a Node which can be evaluated

Implementation: NodeBooleanExpression.java

5.2.2.6.1 NodeComparator

Quite the same as NodeBooleanExpression

JSON format of a *NodeComparator*:

```
{
	"type": "comparator",
	"operator": "==",
	"leftOperand": {},
	"rightOperand": {}
}
```

Implementation: NodeComparator.java

5.2.2.7 NodeWhen

JSON format of a *NodeWhen*:

```
{
	"type": "when",
	"events": {
	},
	"seqRulesThen": {
	}
}
```

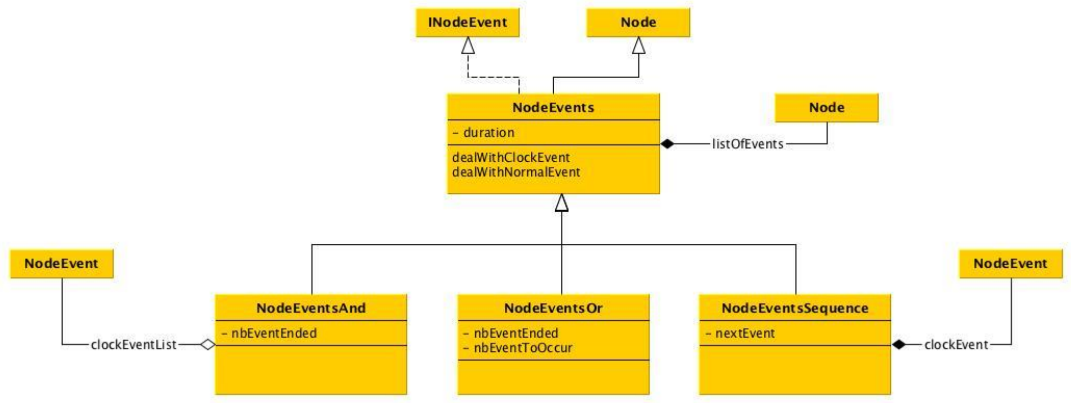
Where:

- *type* has always the value *when*
- *seqEvents* is a *NodeSeqEvent* stored in its JSON format.
- *seqRulesThen* is the sequence of rules to interpret when all the events have been caught. It is a *NodeSeqRules* in its JSON representation.

Implementation: NodeWhen.java

5.2.2.8NodeEvents

This Node class is here to represent either a sequence of events, or a choice between different events.



5.2.2.8.1 NodeEventsOr

JSON format of a *NodeEventsOr*:

```
{
      "type": "eventsOr",
      "duration": 10,
      "events": [],
      "nbEventToOccur" : 10

}
```

Where:

- *type* is always eventsOr
- *duration* in seconds and is an optional parameter
- *events* is an array of event
- *nbEventsToOccur* the number of event that have to happen before the events is triggered

Implementation: NodeEventsOr.java

5.2.2.8.2 NodeEventsAnd

JSON format of a *NodeEventsAnd*:

```
{
      "type": "eventsAnd",
      "duration": ,
      "events": []

}
```

Where:

- *type* is always eventsAnd
- *duration* in seconds and optional
- *events* is an array of event

Implementation: NodeEventsAnd.java

5.2.2.8.3 NodeEventsSequence

JSON format of a *NodeEventsSequence*:

```
{
      "type": "eventsSequence",
      "duration": ,
      "events": []

}
```

Where:

- *type* is always eventsSequence
- *duration* in seconds and optional
- *events* is an array of event

Implementation: NodeEventsSequence.java

5.2.2.9 NodeEvent

JSON format of a *NodeEvent*:

```
{
      "type": "event",
      "eventName": "switchNumber",
      "eventValue": "1",
      "source": {}
}
```

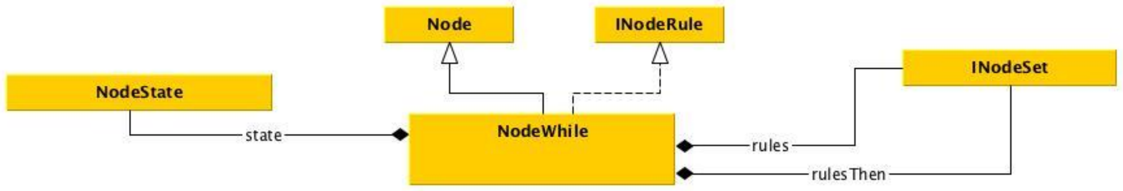
Where:

- *type* is always event
- *source*
  - *type* can be device or program
  - *value* is the device or program ID to listen to.
- *eventName* is the name of the event to listen to.

- *eventValue* is the value to wait for to validate the node. If the node is not interested in a particular value, *eventValue* is an empty string.

Implementation: NodeEvent.java

5.2.2.10 NodeWhile



JSON format of a *NodeWhile*:

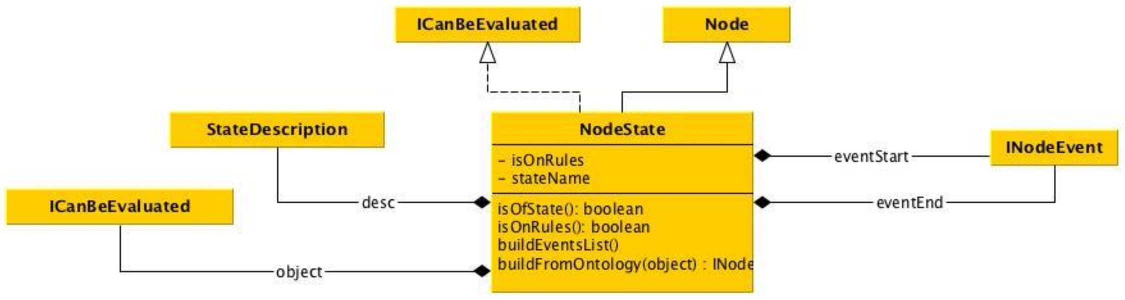
```
{
      "type": "while",
      "state": {},
      "rules": {},
      "rulesThen": {}
}
```

Where:

- *type* is always while
- *state* is the state node description (in JSON)
- *rules* the rules to execute while the state is true (ie keepstate).
- *rulesThen* the rules to execute once the state become false.

Implementation: NodeWhile.java

5.2.2.11 NodeState



JSON format of a *NodeState*:

```
{
      "type": "state",
      "object": {“type”:”device”,
                      “value”:”theDeviceId”
                      },
      "stateName": “aName”
}
```

Where:

- *type* is always while
- *object* is the description of the device
    - *type* will be device
    - *value* is the id of the device
- *stateName* is the name of the state as described in *grammar.json* file.

Implementation: NodeState.java

5.3 The SPOK Editor

In order to support a dynamically extensible grammar as well as to provide end-users with feedforward at the user interface of the editor, the grammar used by the editor is split into 2 parts: the root grammar and the device specific grammars.

The root grammar specifies the generic structures of an end-user program: loops, conditions, etc. It is described in: grammar.peg. The device specific grammars are separated from the root grammar to be able to dynamically build the final grammar to be compliant with what is currently installed and detected by the AppsGate server.

Each device type brings with it its own events, status and actions. To do this, each device must provide some function when it is described.

5.3.1 Parser generation

The goal of the parser is to compile end-user programs into the JSON format described in the previous section. As mentioned in Section 1, the parser relies on PEG.js. PEG.js takes a grammar as input expressed in the format described on PEG online documentation (Grammar format for PEG.js).

5.3.2 User support and compilation

End-users enter syntactic units by pressing any button of a Smart keyboard shown on the screen. The set of buttons and their label correspond to the terminals of the grammar. To achieve this, the end-user program under construction is analyzed by the parser after each button press. Two situations may occur:

1– **The program is not syntactically correct.** The parser throws an exception that specifies the set of the next possible correct terminals. The exception is caught by the editor and the set of buttons is updated according to the possibilities described in the exception.

2– **The program is syntactically correct.** The parser has succeeded to build the AST of the program. The AST is stored in a JSON format, ready to be sent to the AppsGate server. To be able to show the next possibilities in this case, an error is inserted at the end of the program and a new compilation is launched. Now, an exception is thrown with the possibilities that are expected next.

The class diagrams of Figure 13 show the components of the Program module. According to the Backbone architectural decomposition, an end-user program is represented by:

- a ProgramModel which maintains the state (e.g., running) as well as the body (i.e. the AST) of the end-user program. The body is a copy of the program when saved by the end-user,
- a number of views: a ProgramListView which displays the list of all existing end-user programs, a ProgramReaderView which displays the program as a read only presentation, and a ProgramEditorView that allows the end-user to modify the program,
- a ProgramRouter as a dialogue controller to manage navigation between the program views.

The Mediator component contains the current AST of the program in the JSON format. This representation is initialized with a copy of the JSON body from the ProgramModel. The Mediator is subdivided into two subcompoents: the ProgramEditAreaBuilder and the ProgramKeyboardBuilder. The ProgramEditAreaBuilder builds an html representation of the program, using the template associated to each node type of the program. Depending on the syntactic correctness of the program as well as on the node currently selected, the JSON representation of the program is then parsed using PEG.js and the new expected elements are identified. For each of these expected elements, using the associated templates, HTML nodes are built by the ProgramKeyboardBuilder and organized into the Smart Keyboard using CSS classes.

The sequence diagram of Figure 14 illustrates the communication between the AppsGate client components involved in displaying a program.

Figure 14 represents the components involved when a user has pressed a button of the Smart Keyboard. Each button of the Smart Keyboard contains a representation of the program node it stands for in the JSON format. When the user presses a button, the node is added to the JSON representation of the program at the selected position. An event is then triggered in order to update the HTML representation of the program and of the Smart Keyboard. Note that the interpreter (which runs on the AppsGate server), is not incrementally notified of the modifications of a program. It is notified when the end-user is done with program editing.

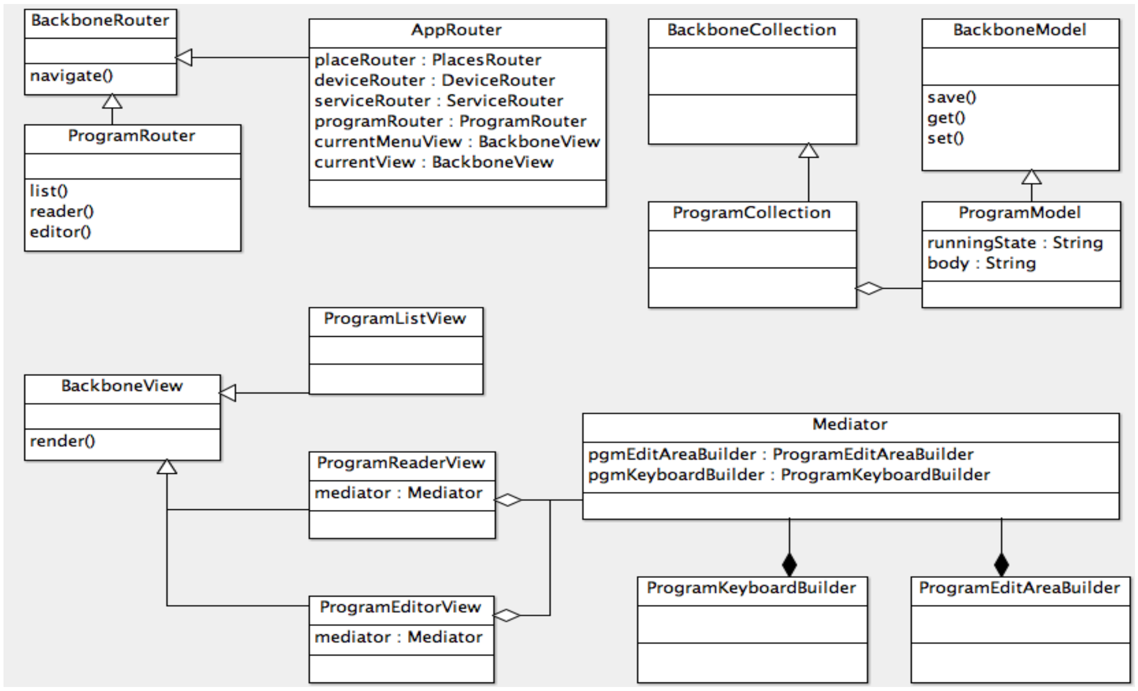


Figure 13. The components of the Program Module.

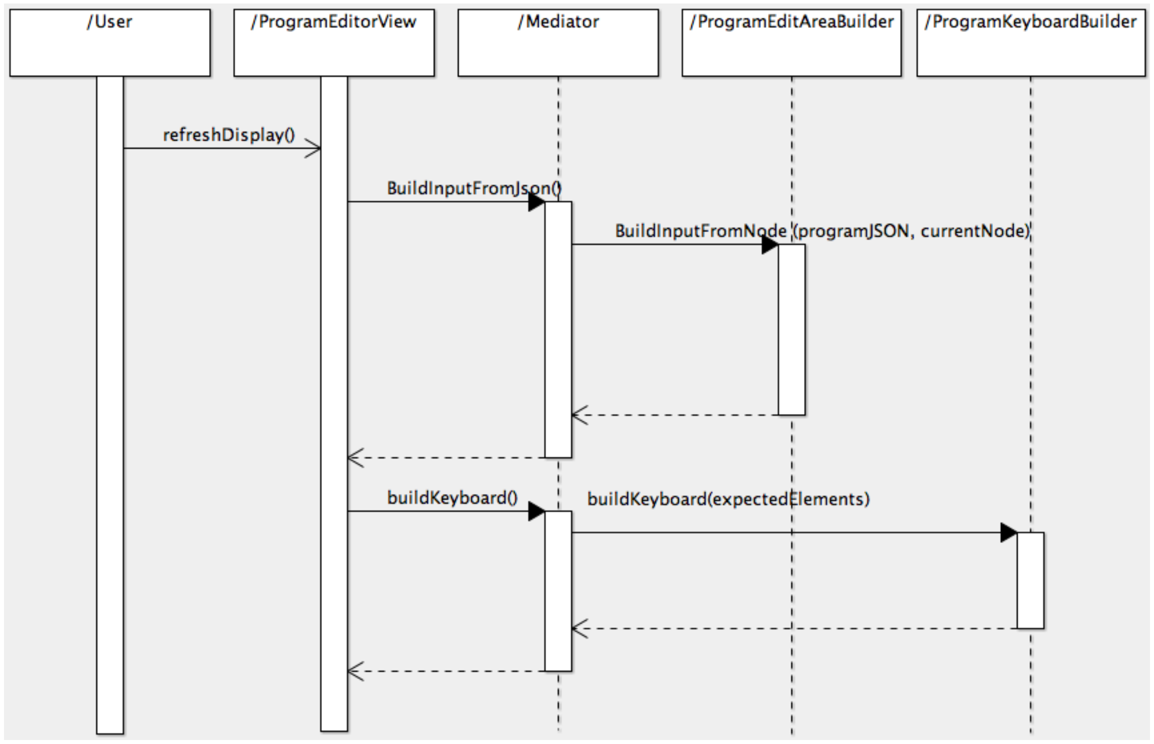


Figure 14. Refreshing a program editor view

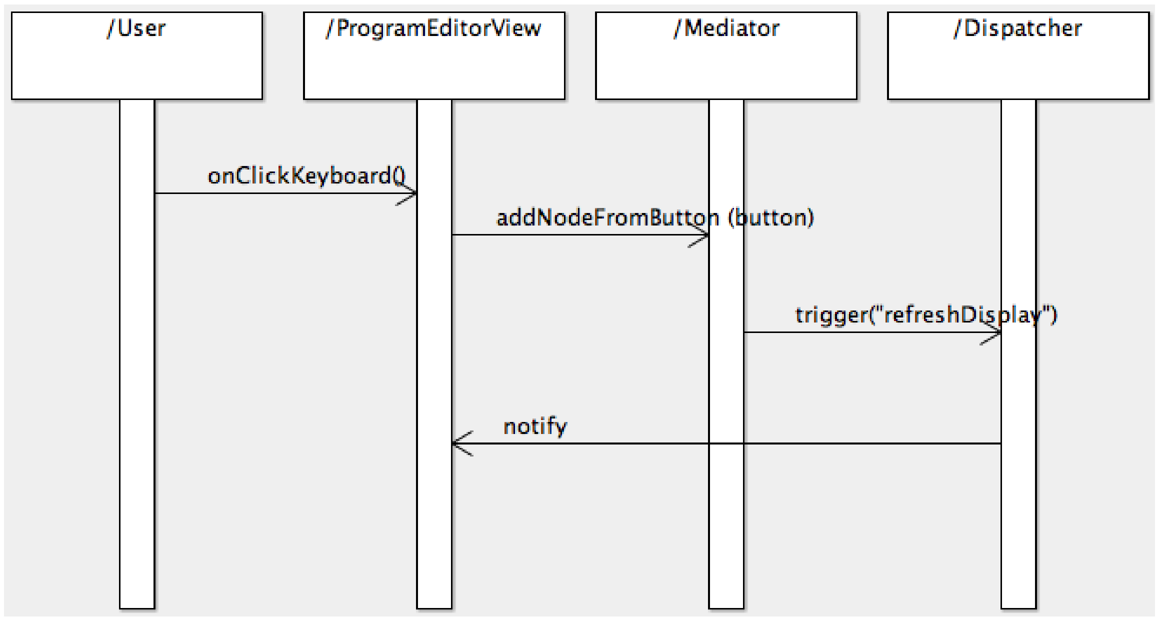


Figure 15. Key press of the Smart Keyboard **by the end-user**