Interactive Spaces Documentation

Release 1.5.4

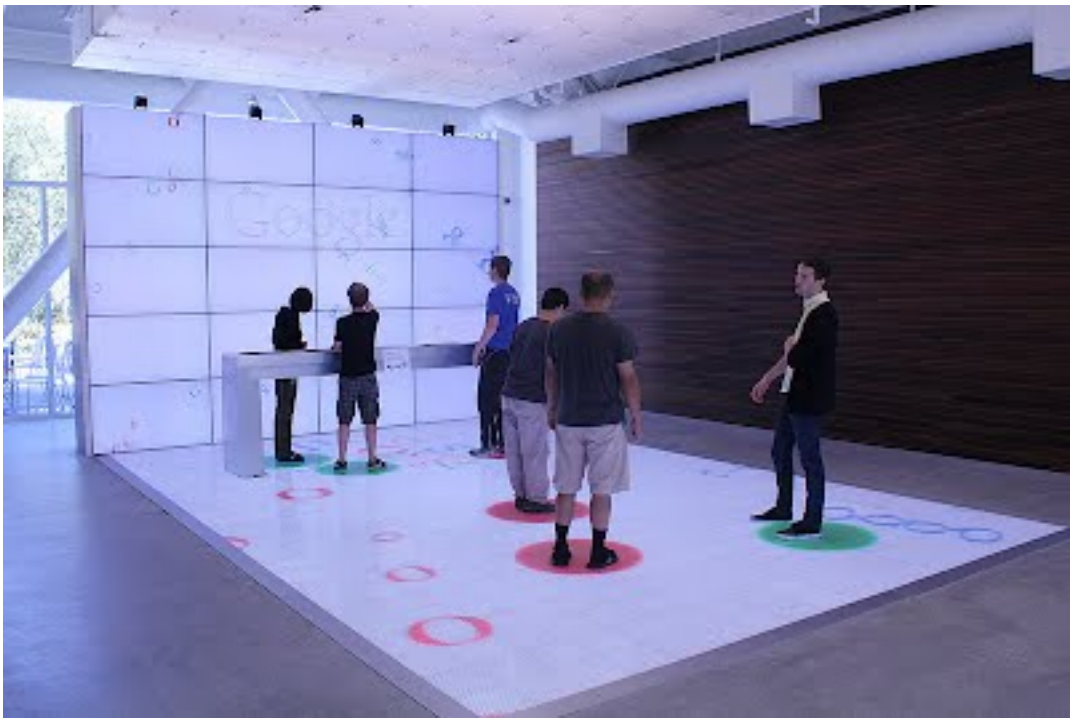
khughes@google.com (Keith Hughes)

December 20, 2013

CONTENTS

Interactive Spaces are physical spaces which can interact with their occupants in some way.

An example can be seen in the picture below.



Here there are cameras in the ceiling which are doing blob tracking, the blobs in this case are people walking around on the floor. The floor then responds by having a colored circle appear underneath the person, following them around as they move around the floor.

You can think of the space as having event producers and consumers. Event producers, things like push buttons, keyboards, and pressure, and proximity sensors, can tell something about the space's occupants and what they are doing. Event consumers then respond to those event and could do anything from putting something on a video screen, moving a physical arm, or turning on a light.

The Interactive Spaces framework is both a collection of libraries for writing activities in the physical space and a runtime environment which will run these activities and allow you to control them. The system has been designed so that you can start with relatively simple activities where Interactive Spaces is doing most of the work, but can get as advanced as you want. Hopefully peeling that onion won't bring too many tears to your eyes.

CHAPTER

ONE

HISTORY AND DESIGN PHILOSOPHY

1.1 History

Interactive Spaces (IS) was created for an Experience Center on the Mountain View Campus of Google, which was to contain interactive displays. IS is the software glue that holds the activities in the Center together. It was started in 2011 and the Center went live in 2012.

The interactive displays in the Center were a collaboration between Google and the Rockwell Group's LAB division. Google wrote the IS code. Rockwell wrote the activities which run in the Center using a combination of the Interactive Spaces APIs and also wrote some of the software for some of the pieces that weren't supported by IS for the initial release of that project.

The initial implementation didn't have everything envisioned for the project, but contained enough of an API for all of the initial installations planned for the Center and many of the pieces needed to run a production installation.

1.2 Design Philosophy

Interactive Spaces was designed with several concepts in mind:

- First, it needed to run a production Center like the one in Mountain View. So it had to support deployment, control, maintenance, and monitoring of applications which run in an interactive space, much like a small data center.
- It had to be as portable as possible, hence the choice of Java.
- It should be able to leverage code meant for controlling physical devices from Centers, part of the reason that it was decided that the main communication protocol should be the Robot Operating System (ROS).
- It should be easy to reconfigure and redeploy an entire space quickly, part of the reason for the use of OSGi.
- The learning curve of IS should be shallow in the beginning and gradually get steeper as a developer needs more functionality. It should be an onion, where the developer keeps peeling layers away until they find the level where they have the power they need.
- IS should provide as many of the sorts of functionality that an interactive designer and developer would need out of the box. Hence it comes with web servers, easy to use thread pools, remote communication, job schedulers, and a whole host of other functionality.
- It should be possible to implement activities in languages other than Java, and have some sort of control over activities which are not implemented as IS activities.

CHAPTER

TWO

INSTALLING AND UPDATING INTERACTIVE SPACES

Before we can get started, we need to install Interactive spaces on a computer. Installing and updating Interactive Spaces is pretty simple.

2.1 Installing Interactive Spaces

Installing Interactive Spaces from the supplied installers is very easy.

2.1.1 Prerequisites

Before you can install Interactive Spaces on your computer, you should make sure you have Java installed first.

Interactive Spaces requires at least Java 1.6.

If you want to be able to easily build activities, you should install the Ant building tool. This is going away for non-Java activities and possibly even for Java activities.

2.1.2 Installing a Local Master

Installing a master is pretty easy. You will run an installer activity, and finally test your installation.

Installing the Master

If you are using a windowing system, find the icon for the Interactive Spaces Master installer and double click on it. The installer is a Java jar file.

If you are using a command line interface for your operating system, use the command

```
java -jar interactivespaces-master-installer-x.y.z-standard.jar
```

where *x.y.z* is the version of the Interactive Spaces Master you are installing.

For now just accept all of the default settings by clicking Next on the configuration page.

Testing the Master Installation

To test if your installation happened correctly, open up a command shell and go to the directory where you installed the master. Once there, type the command

```
bin/startup_linux.bash
```

You should see a bunch of text scroll by as the master starts up. When you see no more text going by, go to a web browser and go to

```
http://localhost:8080/interactivespaces
```

If everything installed correctly you should be seeing the Master Web Interface in your browser.

2.1.3 Installing a Local Controller

Installing a controller is pretty easy if you chose to let the controller autoconfigure itself. You will run an installer activity, and finally test your installation.

You can also manually configure the controller, though there usually isn't a good reason for this.

Installing the Controller

If you are using a windowing system, find the icon for the Interactive Spaces Controller installer and double click on it. The installer is a Java jar file.

If you are using a command line interface for your operating system, use the command

```
java -jar interactivespaces-controller-installer-x.y.z-standard.jar
```

where *x.y.z* is the version of the Interactive Spaces Controller you are installing.

If you are auto-configuring the controller, make sure you don't check the manual configuration checkbox.

Testing the Controller Installation

To test if your installation happened correctly, first make sure you have a Master started. Then open up a command shell and go to the directory where you installed the controller. Once there, type the command

```
bin/startup_linux.bash
```

You should see a bunch of text scroll by as the controller starts up.

Go to the Master Web Interface in your browser. The URL is

```
http://localhost:8080/interactivespaces
```

Go to the Controller menu. You should see an entry with the name of the controller you created. Click on this and click Connect. If everything is working you should see

```
New subscriber for controller status
```

appear in the controller window. Also, if you refresh the controller page in the Master Web Interface you should see it say that the controller is in the running state.

Manually Configuring a Controller

This is an advanced topic and is not normally done. Only do this if you really have no other choice.

Before you install a manually configured controller, you need a UUID for the controller.

You can get this by creating a new controller in the Master webapp. Click on the **Space Controller** menu, then *New*. Decide on a Host ID for the controller. The Controller Name you use is only for the master, pick something descriptive for the controller. Then click *Save*. The master will create a UUID for the controller and display it in the next screen.

You will enter both the controller Host ID and UUID during the controller installation when prompted. Be sure to chose the manually configured option during installation.

2.1.4 Installing the Workbench

The Interactive Spaces Workbench provides you with example code, documentation, and the Workbench application which can help you maintain and deploy your activities.

If you are using a windowing system, find the icon for the Interactive Spaces Controller installer and double click on it. The installer is a Java jar file.

If you are using a command line interface for your operating system, use the command

```
java -jar interactivespaces-workbench-installer-x.y.z-standard.jar
```

where *x.y.z* is the version of the Interactive Spaces Workbench you are installing.

2.2 Updating An Installation

Updating an installation is currently more complicated than it needs to be, but that will change soon.

Delete the contents of the following folders in your master and in all of your controllers and workbenches.

1. bootstrap
2. lib/system/java

Also delete the file *interactivespaces-launcher-x.y.z* from the root folder of each master, controller, and workbench, where *x.y.z* was the version of the launcher that was there before the update.

Once you have done this, you can upgrade the same way you install.

CHAPTER

THREE

YOUR FIRST INTERACTIVE SPACES ACTIVITIES

Let's get your first Interactive Spaces activity installed and running. To keep it simple, you will install one of the sample activities that comes with an Interactive Spaces Workbench installation. After that you will create an activity from scratch.

Make sure you start the Master up first and then your Controller. When you shut things down, you should shut the Controller down first, then the Master. To shut them down, go into the shell window where you started them up and type Ctrl-D, where Ctrl is the control key on your keyboard.

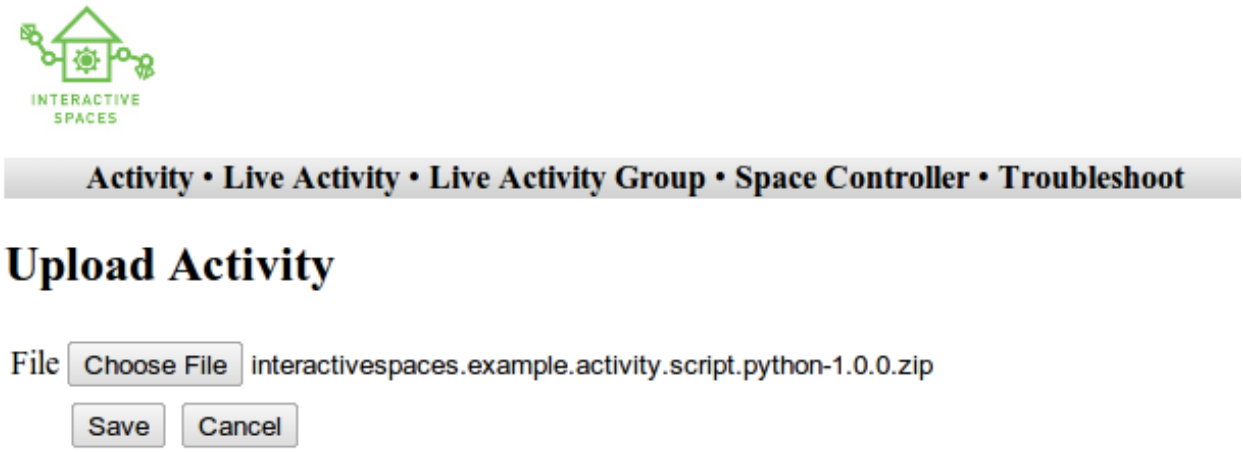
3.1 Running An Sample Activity

Let's start off by uploading one of the sample activities found in the Interactive Spaces Workbench into Interactive Spaces. This will help demonstrate some of the basic Interactive Spaces concepts.

3.1.1 Uploading The Activity

The first step is to load the sample activity into the Master. To do this, go to the Master Web Interface and click on the **Activity** menu, then **Upload**. This will show you the following screen.

Click the **Choose File** button and go to where you installed the Interactive Spaces Workbench. Find the `interactivespaces.example.activity.hello` in the `examples/basics/hello` folder of the Workbench, go into the target folder, and select the file ending with .zip. You should end up with something like the following.



From here, click **Save**. You should then see

Activity • Live Activity • Live Activity Group • Space Controller • Troubleshoot

Activity: Python Script Activity Example

Edit Delete

An example of an Interactive Spaces Activity written in Python.

Version	1.0.0
Identifying Name	interactivespaces.example.activity.script.python
Last Uploaded	Mar 15, 2012 11:35:44 AM

Installed Activities

None

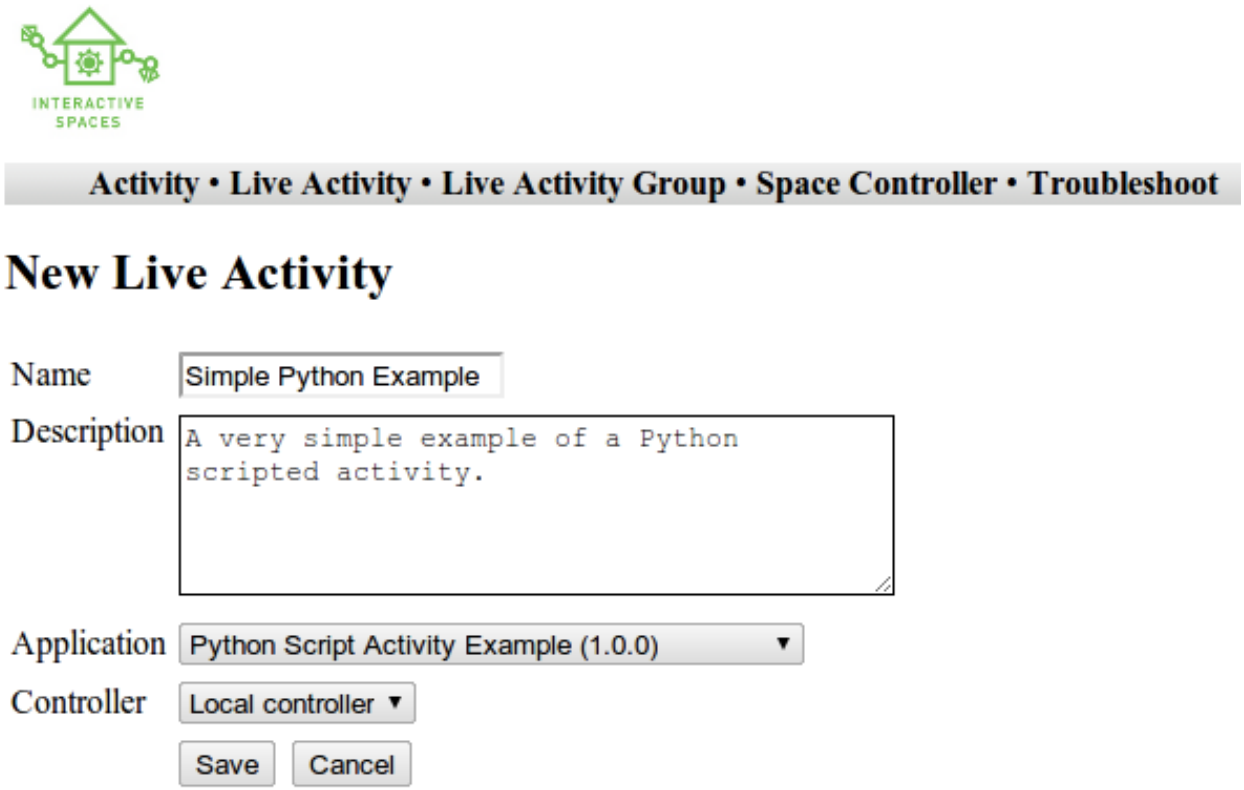
Dependencies

None

3.1.2 Creating the Live Activity

Before you can deploy and run an Activity, you must first create an Live Activity. This allows you to say which Controller the Activity will be deployed on.

Click Live Activity and then **New**. This will give you the following screen.



Since this is the first activity you are installing, you should only have one Controller and One Activity to choose from the appropriate dropdowns. So pick a descriptive name for your activity and write a short description and hit **Save**. This will take you to the following screen.

3.1.3 Deploying the Live Activity

You need to deploy the Live Activity to the Controller before you can run it. To do this, click on the **Deploy** button on the Live Activity Screen. If you look in the Master and Controller consoles, you should see some logging message about deploying the activity. The logs will use the UUID you see in the example above, in my installation it was `67d4232c-b61b-46cd-ae20-c0834f9e972d`, yours will be different.

3.1.4 Starting Up The Live Activity

You can now start the activity up by clicking the **Startup** command. You should see the startup happen in the logs in both the Master and Controller consoles, once again by using the UUID to identify the Live Activity. You should also see an error log in the Controller Console saying that the Live Activity has started up. This message is coming from the Python script, which you can see in the Workbench examples folder.

3.1.5 Activating The Live Activity

Live Activities must be Activated before they can handle any requests. You will find out more about what that means later. For now you can now activate the activity up by clicking the **Activate** command. You should see the activation happen in the logs in both the Master and Controller consoles, including an error log in the Controller Console from the Python script saying that the Live Activity has activated.

3.1.6 Deactivating The Live Activity

Activated Live Activities can be either Deactivated if you want them to stop processing requests but keep running, or Shutdown. You can now deactivate the activity up by clicking the **Deactivate** command. You should see the deactivation happen in the logs in both the Master and Controller consoles, including an error log in the Controller Console from the Python script saying that the Live Activity has deactivated.

3.1.7 Shutting Down The Live Activity

Shut down the activity up by clicking the **Shutdown** command. You should see the shutdown happen in the logs in both the Master and Controller consoles, including an error log in the Controller Console from the Python script saying that the Live Activity has shut down.

3.2 Creating an Activity Project From Scratch

The Workbench provides a bunch of operations for working with activities, including the ability to create new projects and also build them.

3.2.1 Creating the Activity Project

First let's create a new Activity project.

You can easily create template projects in Java, Python, and Javascript. Let's start off with a Java project.

From the command line go to the directory where you installed the Workbench. Once there, type the following command.

```
bin/isworkbench.bash create language java
```

You will then be asked a series of questions. Let's create a project called *me.activity.first*.

The first question is the identifying name.

```
Identifying name:
```

This, along with the version given later uniquely identifies the Activity to Interactive Spaces.

```
Identifying name: me.activity.first
```

The identifying name is is a dot separated set of names, Examples would be things like

- a.b.c
- com.google.myactivity

Each part of the name must start with a letter and can then be letters, digits, and underscores.

Next is the version.

```
Version:
```

Let's make it version 1.

```
Version: 1.0.0
```

Versions consists of 3 sets of numbers, separated by dots. Examples would be

- 1.0.0
- 0.1.0-beta

Notice the last one has a dash followed by some text.

Next is the name.

```
Name:
```

Let's use the name *My First Interactive Spaces Activity*.

```
Name: My First Interactive Spaces Activity
```

The name is the human readable name which shows up in the Interactive Spaces web UI.

Finally there is a description.

```
Description:
```

Make this description anything you want.

This field is not required. It is a description of the activity.

The project will then be created in the Workbench directory.

3.2.2 Building the Activity

The activity is built using the Workbench.

Suppose the name of your project is *me.activity.first*. The command to build your project is

```
bin/isworkbench.bash me.activity.first build
```

This will put a file called *me.activity.first-1.0.0.zip* in the *me.activity.first/build* directory.

You can upload this activity into Interactive Spaces the same way you did the sample activity.

3.2.3 Using an IDE

You can create an IDE project for your activity projects, even if they aren't Java projects.

Suppose the name of your project is *me.activity.first*. The command to create the IDE project is

```
bin/isworkbench.bash me.activity.first ide eclipse
```

This will build an Eclipse project which you can then import into Eclipse.

3.3 Next Steps

You have now walked through installing an activity on a controller and running it. You should look at the various examples in the Interactive Spaces Workbench to get an idea of the types of activities you can create in Interactive Spaces.

In the next chapter we will examine the basics of Interactive Spaces in more detail.

CHAPTER

FOUR

INTERACTIVE SPACES BASICS

Now that you have installed and run your first Interactive Spaces activity it is time to understand more of the pieces of what you have done. In this section we will cover all of the basic concepts of Interactive Spaces. While reading, think back to what you did in the previous chapter.

4.1 Overview

An "interactive space", in Interactive Spaces terms, is a collection of event producers and consumers in a physical space which can sense what is happening inside the space and then react to it in some manner.

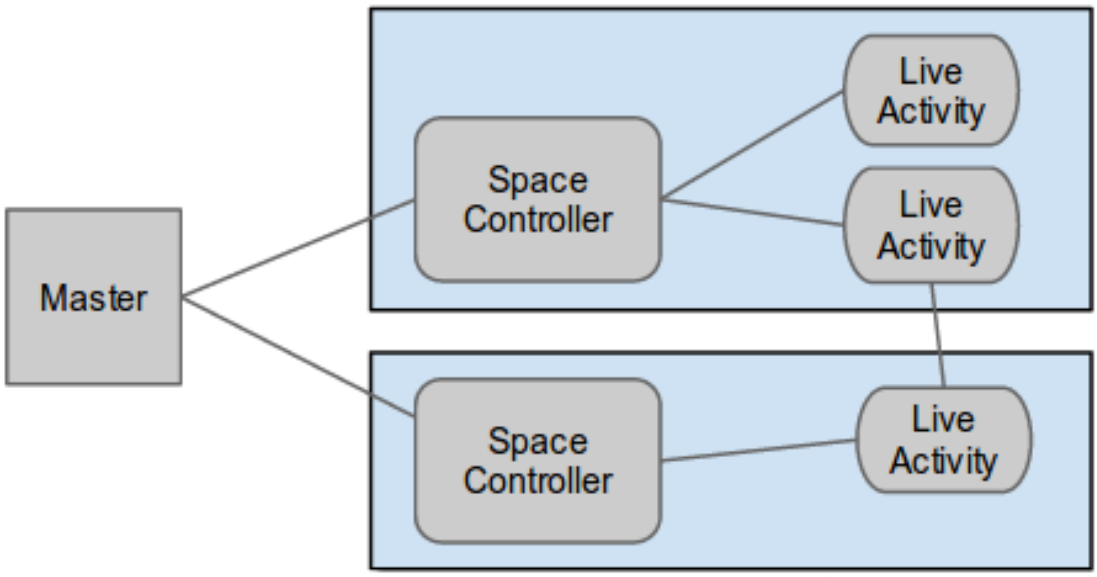
A simple example of an event producer would be a pressure sensor under the carpet signaling that someone has stepped on it. A more complex event might include the use of a depth camera whose events would give the angles of all the joints of the person it is tracking.

A simple example of an event consumer would be a light which turns on, while a more complex one might be speech synthesis being used to tell you to not step on the carpet.

These event consumers and producers can then be connected in interesting ways. The events used by gesture recognition could be used to both turn on the light and have the speech synthesis machine tell you it is impolite to point.

The event producers and consumers in Interactive Spaces are implemented as Live Activities. A Live Activity is some computer program running somewhere in the space which is either producing events or consuming events, or even both at the same time.

The following diagram will help to understand how all the pieces are put together.



Those previously mentioned Live Activities need to run on a computer somewhere in the space. In Interactive Spaces, a Space Controller is the container for Live Activities which runs on a given computer. A given Space Controller can run many Live Activities as need to run on that machine. Though there is nothing to stop you from running multiple Space Controllers on a given machine, there is rarely, if ever, a reason to. Live Activities need a Space Controller to run in. The Controller directly controls the Live Activities it contains and tells them to start, stop, and several other operations to be described later. In the diagram above, the large rectangles which contain a Space Controller and some Live Activities are the computers. The diagram shows two computers.

Live Activities are controlled remotely by the Master. The Master knows about all Space Controllers in the space and what Live Activities are on each Controller. If you want to deploy a Live Activity, update it, start it running, shut it down, or any of a number of other operations, this is all done from the Master which tells the Space Controller holding the Live Activity what it wants the Live Activity to do. You can also control Space Controller-specific operations from the Master, such as telling the Controller to shut down all Live Activities currently running on it and/or shut itself down.

A given installation of Interactive Spaces can run on a single computer or an entire network of computers. You can run both a Master and a Space Controller on the same machine, in fact, this is often done to give yourself a development environment, but usually you will have a lot of machines talking with each other.

To summarize, Live Activities provide the functionality and interactivity of the space. The Master controls the entire space by speaking remotely to the Space Controllers which contain and directly control the Live Activities. The Live Activities produce and consume events happening inside the space by speaking directly with each other.

4.2 Activities and Live Activities

Live Activities are the work horses of an interactive space, without them the space would have no interactivity. They contain an Activity and potentially a configuration. They also have a lifecycle.

Let's look at each of these.

4.2.1 Activities and Configurations

Live Activities are versions of Activities which are installed on a given Space Controller. A good analogy for an Activity is a program you have an install disk for, say a graphics program, and the Live Activities are the copies of that graphics program that you have installed on both your laptop and on your desktop. You can think of the laptop and desktop in this example as the Space Controllers running your graphics program Live Activities.

Activities come with base configurations which give default values for different parameters which can control how the Live Activity works. Each Live Activity can also have its own configuration which can override any of the values found in the configuration which is part of the Activity the Live Activity is based on. In the graphics program example, perhaps on your laptop you have it configured to use the touchpad for drawing, whereas the desktop uses an pen-based active tablet.

The Activity can also contain any initial data that the Live Activity would need to run.

You can have more than one Live Activity based on a given Activity, even on the same Space Controller, each one potentially configured differently from any of the other copies of the Activity in your space. So a Live Activity has potentially 3 parts, it has an Activity, which is the program the Live Activity runs, it has a Space Controller, which specifies which machine the Live Activity runs on. And it potentially has a configuration which makes it slightly different than any of the other Live Activities based on the same underlying Activity.

4.2.2 The Live Activity Lifecycle

Live Activities have a lifecycle, which says what state they are in at any given time. The full lifecycle has a lot of different states to it, for now we will look at the major ones.

READY

: The Live Activity is deployed within the controller and is ready to run.

RUNNING

: The Live Activity is loaded and running, but can't necessarily handle requests.

ACTIVATE

: The Live Activity can handle requests.

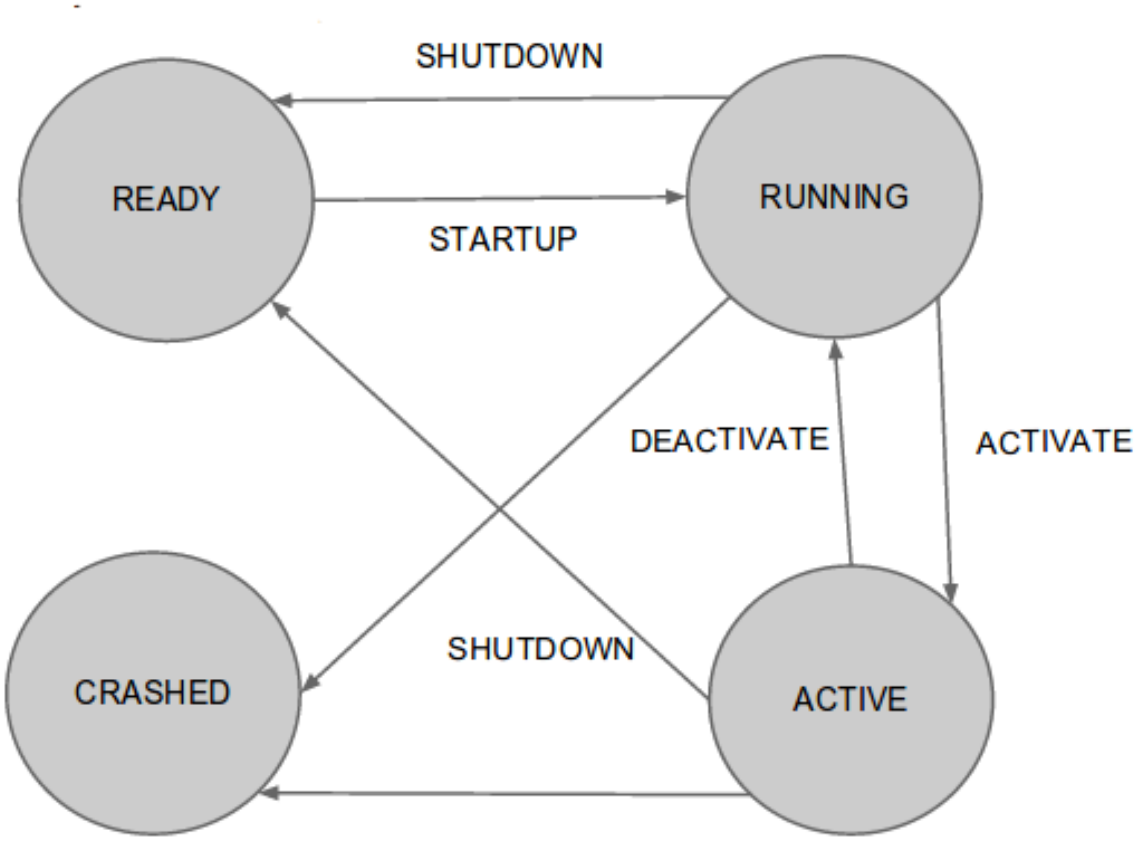
  This is needed because some Live Activities can take a long time to reach the RUNNING state. For example, a piece of hardware such as a projector may take a long time to warm up.

CRASHED

: The Live Activity has crashed and is no longer running.

The Live Activity will move between these states when they are given commands to move to the new state. For instance, if the Live Activity is in the READY state, it can be sent a StartUp command which will make it move to either the RUNNING state, if it was able to start, or CRASHED if it wasn't able to start properly. If the Live Activity is sent an Activate command, it will transfer from RUNNING to ACTIVATED.

This diagram may help explain all of the main states and transitions.



A complete list of the states and what they mean will be found in the Advanced Activity section.

4.3 Space Controllers

As we've already seen, Space Controllers are where Live Activities run. if you have an interactive space, you need at least 1 Space Controller because you have no space without Live Activities.

A space will have anywhere from 1 Space Controller to potentially hundreds or thousands, depending on how complex the space is. 99.99999% of the time there will be one Space Controller to one computer. Though nothing prevents multiple Space Controllers running on the same machine, it isn't very likely.

The Space Controller is the container that runs Live Activities. The Space Controller tells the Live Activities it contains when to start, stop, activate, and deactivate. When Live Activities are deployed, the Space Controller copies the Activity from the Activity Repository and unpackages it for installation.

The Space Controller also provides services that Live Activities need to run. The Space Controller knows which operating system it is running on and can make decisions for the Live Activity on which executable to use for a native activity, including a web browser being used by the Live Activity. It provides per-Live Activity logging. The Space Controller also contains services which can be used by multiple Live Activities, like a scripting service or a service for scheduling events in the future, or clocks which can be synchronized across the space. There are many services available, more than can be described here and more coming all the time.

The Space Controller monitors all Live Activities by periodically asking all Live Activities what their state is. The Space Controller uses this information to provide an alerting mechanisms for when Live Activities fail. The Controller

also automatically tries to restart Live Activities which have crashed.

The Master, discussed next, does not communicate directly with Live Activities. The Master communicates with the Space Controller and the Space Controller directly controls the Live Activities.

4.4 The Master

The Interactive Spaces Master is in charge of the entire physical space and is used to not only control the Live Activities (via the Space Controller which contains the Live Activity), but to also support maintenance, deployment, and monitoring of the entire installation.

It would be very interesting to have an interactive space which has much more decentralized control, but Interactive Spaces was built to support installations where it was very easy to tear a space down and reconfigure it for a very different purpose in a short time and this is much easier from a central control point. The Live Activities themselves could be more self organizing, it would be possible to write code where a Live Activity can query about its environment and make functionality available based on what it finds, but the Master is still needed in Interactive Space's view of a space.

The Master contains a model of the entire space. It knows how to contact every Space Controller in the entire space and what Live Activities are supposed to be on that Space Controller. It also knows what Activity a particular Live Activity is based on and whether or not the Space Controller has the most recent version of the Live Activity. It also contains the current configurations for all Live Activities.

The Master contains the Activity Repository which contains all Activities known in the space. The Master is used to deploy a new Live Activity, or new version of an already installed Live Activity, to its Controller and takes the Activity to deploy from this Repository.

The master is also used to start, activate, deactivate, and stop Live Activities. The Space Controllers constantly inform the master about the status of all Live Activities running in the space making it possible from one central location to know everything that is happening in the space.

The Master also allows easy control of a Space Controller. From the Master you can shut down all Live Activities running on the Controller or shut the controller down itself. You can also ask the Controller to immediately give the current state of all Live Activities on the Controller.

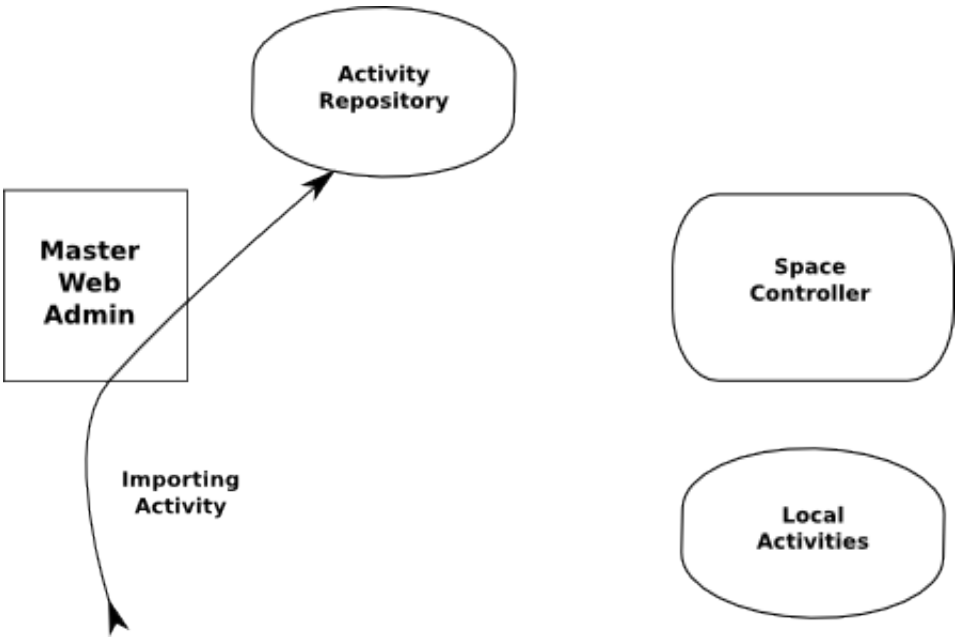
4.4.1 The Master and Live Activity Communication

The Master is also necessary for the communication between Live Activities and between the Master and the Space Controllers. Interactive Spaces communication works by having global topics that can be listened to or written to. The Master provides a global registry for all of these topics. Each topic contains information about who wants to listen to what is written on the topics and who wants to write on the topics.

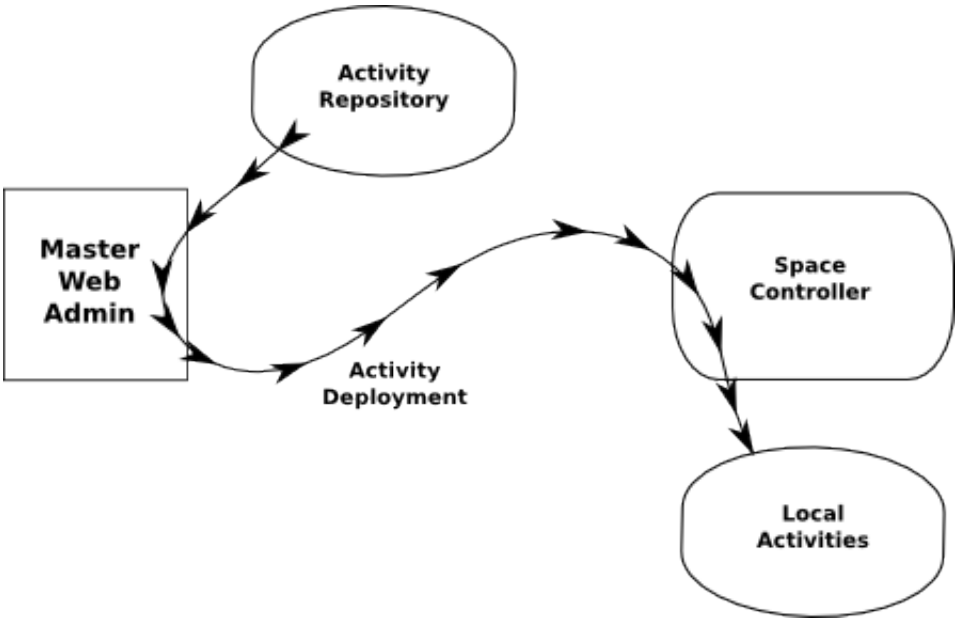
As an example, there might be a topic in the space called */livingroom/camera*. The camera itself would write information on this topic about what it is seeing. Listeners would listen to this topic and process the images that are being seen. The Master would have a record for */livingroom/camera* with all of the publishers of the events on that topic, probably just the camera Live Activity in this case, and consumers of the camera events.

4.4.2 Importing and Deploying Activities

Importing an Activity into the Master places the Activity in the Activity Repository and places information about the Activity into the Master Domain Model.



Deploying a Live Activity from the Master involves copying the Activity that the Live Activity is based on from the Master Activity Repository to the Local Activities for the Space Controller that hosts the Live Activity.



Deploying does not copy any configurations associated with the Live Activity, it is necessary to Configure the Live Activity from the Master Web Admin for the Live Activity configurations to be sent to the Space Controller.

4.5 Live Activity Group

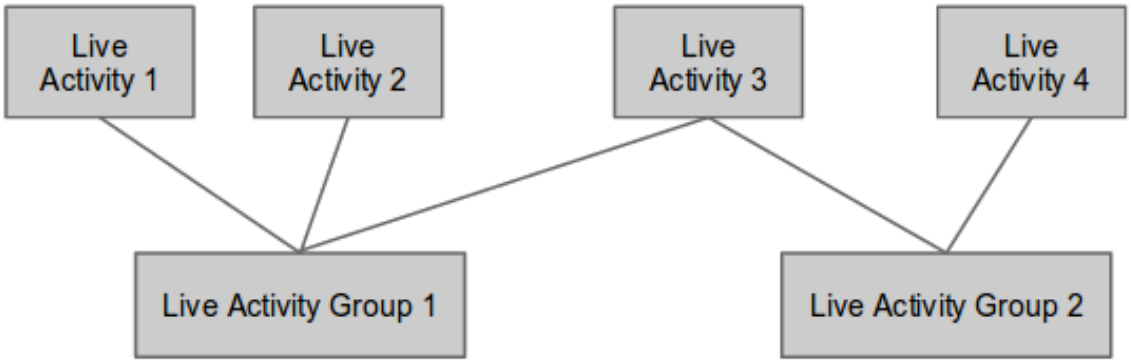
A Live Activity Group is a group of Live Activities which are controlled as a single unit. This is useful because often you will need a collection of event producers and consumers to give a certain behavior in your space. Often these will be implemented as separate Live Activities, but it is best to treat them as a single unit. Groups make this possible.

The Group only has meaning on the Master, Controllers only understand about individual Live Activities.

Groups are deployed by deploying each Live Activity in the Group. They also have the same lifecycle as a Live Activity and can be started, activated, deactivated, and shutdown as a group. the particular lifecycle request will be sent to each Live Activity in the Group.

However, there is one slight difference in how the lifecycle requests are handled.

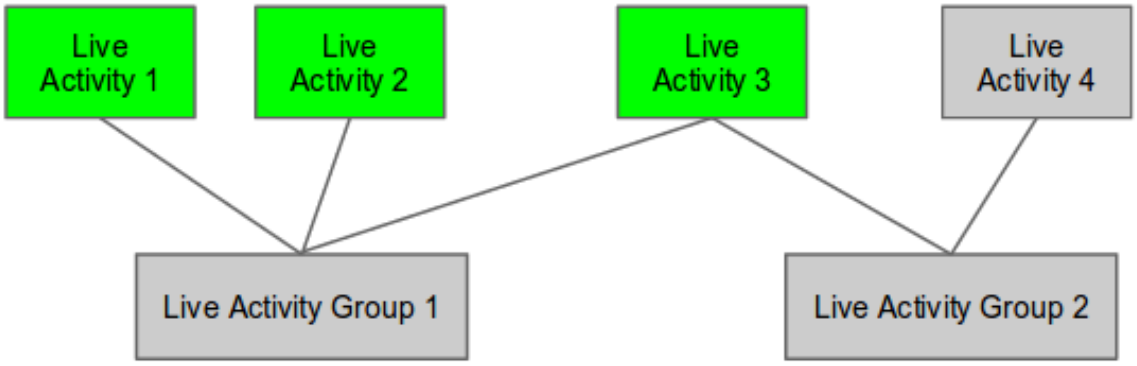
A given Live Activity can be in more than 1 group. Let's say we have two Live Activity Groups, Live Activity Group 1 and Live Activity Group 2.



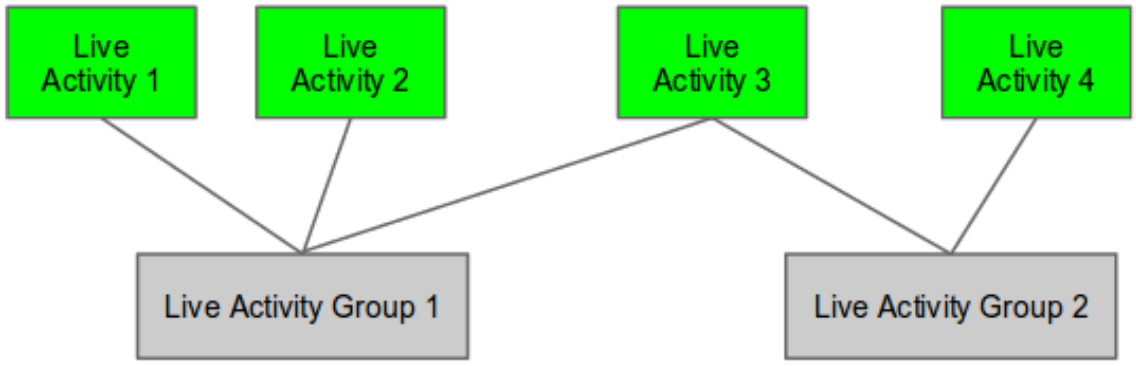
Live Activity Group 1 consists of Live Activity 1, Live Activity 2, and Live Activity 3. Live Activity Group 2 consists of Live Activity 3, and Live Activity 4.

Initially nothing is running in the entire space.

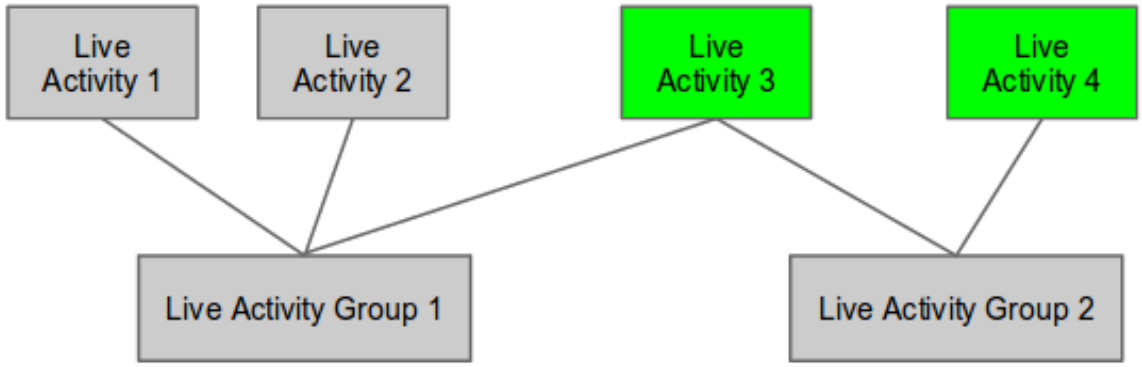
Suppose we start Live Activity Group 1. Because Live Activity 1, Live Activity 2, and Live Activity 3 aren't running, they all start.



Suppose we next start Live Activity Group 2. Live Activity 3 is already running, so there is no need to start it again and it is left alone. But Live Activity 4 is not running yet, so only it will be started.



Suppose we now want to shut Live Activity Group 1 down. We can immediately shutdown Live Activity 1 and Live Activity 2 because they aren't being used anywhere else. But Live Activity 3 is still needed by Live Activity Group 2, so can't be shut down. Live Activity 4 is left running.



So once Live Activities are part of a Live Activity Group and are controlled at the Group level, they will only be started for the first Group which asks them to start, and will only be stopped by the last Group that started them asks them to be shut down.

The same thing happens with activation. The first group which activates the Live Activity will cause it to be activated, but it won't be deactivated until the last remaining Group which activated it asks it to be deactivated.

Live Activities can be in as many Live Activity Groups as is desired.

4.6 Spaces

Suppose you have a physical space that you want to slice and dice in many different ways so that you can refer to items in ways that make sense. For instance, suppose you have a two story house. You might want to refer to all the Live Activities

- on the first floor
- on the second floor.
- the living room on the first floor

- all of the bedrooms as a unit, even though some of them are on the first floor and some are on the second floor
- all of the camera Live Activities in the entire house

Sometimes your slicing has to do with geographic location (the floors of the house, or the living room), sometimes it has to do with function of the space (the bedrooms), and sometimes to do with the functionality (the cameras).

Spaces allow you to to this. Admittedly *space* is not necessarily a good name for referring to all of the cameras as a unit, but it seemed the best term overall.

Spaces consist of two things

- an arbitrary number of Live Activity Groups (including 0)
- an arbitrary number of child Spaces (including 0)

Live Activity Groups can appear in more than one space. So perhaps you have a depth camera halfway up the stairs as a small Live Activity group, that group can be part of the Stair Space, the First Floor Space, and the Second Floor Space. The Living Room will be a Space, and that Space could be a child Space of the First Floor Space.

You can deploy a Space, which means that every Live Activity Group of the Space will be deployed, and every Live Activity Group of all child Spaces and their children until you get to child Spaces that have no children. You can also start, stop, activate and deactivate the Space with the same behavior.

CHAPTER

FIVE

INTERACTIVE SPACES EXAMPLE ACTIVITIES

The Interactive Spaces Workbench contains a variety of example activities in its *examples* folder. These are to provide you with some very basic examples of the sorts of things you can do with Interactive Spaces.

The number of examples will continue to grow as more functionality is placed into Interactive Spaces.

Remember, the artifact you want to upload to the Interactive Spaces Master is the *zip* file found in the *build* folder of the example. For example, to upload the simplest Hello World Activity, load the file *interactivespaces.example.activity.hello-1.0.0.zip* found in the folder *examples/basics/hello/interactivespaces.example.activity.hello/build* in the Workbench install folder.

Beware, sometimes there will also be a *jar* file, do not use that one.

5.1 Hello World: The Simple Events

The following are written in a variety of programming languages and are found in the *examples/basics/hello* folder of the workbench.

- interactivespaces.example.activity.hello
- interactivespaces.example.activity.hello.javascript
- interactivespaces.example.activity.hello.python

These activities merely log some Interactive Spaces events which take place when a Live Activity runs. These examples can help you understand what events happen in what order.

You will need to hit the *Configure* button to see the configuration update event. *Edit Config* should be used first if you have never configured the activity.

If you want less output from these Activities, find *onActivityCheckState* and change *info* to *debug*. This means that checking activity state will only be logged if the logging level for the activity is set to DEBUG.

For fun, there is also a talking Hello World example. Just turn up your computer's speakers and enjoy.

- interactivespaces.example.activity.hello.speech

5.2 Web Activities

You can easily use browser-based applications to provide a UI for your Activities.

The following examples are found in the *examples/basics/web* folder of the Workbench.

5.2.1 Simple Web Activity

It is very easy to use Interactive Spaces to display a web application. Interactive Spaces will provide a web server which will serve your HTML, CSS, and Javascript, and a web browser to display that HTML, CSS, and Javascript.

- interactivespaces.example.activity.hello.web

This is a standalone browser activity which shows a single page and not much else.

Interactive Spaces tries to keep things from crashing. Try closing the browser and see what happens.

5.2.2 Web Socket Activities

The following Activities are browser based, but use Web Sockets so that the Activity code running in the browser can communicate with the Live Activity which is running it on the Space Controller.

- interactivespaces.example.activity.web
- interactivespaces.example.activity.web.javascript
- interactivespaces.example.activity.web.python

Pushing the Activate and Deactivate buttons will show different pictures. Pushing the button in the browser will send a message to the controlling Live Activity which will be printed in the Controller's logs.

The first example is written in Java, the other two are written in Javascript and Python.

5.3 Routable Activities

Interactive Spaces really only become interesting when your space has event producers and event consumers running as their own Activities which speak to each other. only do that on separate controllers.

Interactive Spaces uses ROS for its underlying communication. Usage of ROS can be somewhat intimidating, so the examples below use *routes* which are JSON-based communication over ROS. You can have *input routes* which listen for messages and deliver them to their Live Activities, and *output routes* which write messages out for an input route somewhere to receive. Messages on routes are dictionaries of data which can be turned into JSON messages and read from JSON messages.

The following examples are found in the *examples/basics/comm* folder of the workbench.

- interactivespaces.example.activity.routable.input
- interactivespaces.example.activity.routable.output
- interactivespaces.example.activity.routable.input.python
- interactivespaces.example.activity.routable.output.python

The above output examples will write a message onb a route. The above input examples will read the message and write it in the controller's logs. make sure oyu look in the window where you are running the controller to see the log output.

You need to run both an input activity and an output activity to see these examples work. The first two are written in Java and the second two are written in Python. You can run the pair in the same language or in different languages, it doesn't matter.

Also, the two activities can be on the same controller or on separate controllers. You can also run multiple versions of each activity (such as multiple versions of the input route sample), but then they must run on different controllers.

Want some fun? Run the following route example which will listen on the same route as the examples above, but will speak the message sent over the route rather than just logging it.

- interactivespaces.example.activity.routable.input.speech

It is very easy to have a browser based web application use routes. It requires you to write a Live Activity which will communicate with the browser application over a web socket connection and will listen to routes or write to routes.

- interactivespaces.example.activity.routable.input.web
- interactivespaces.example.activity.routable.output.web

The above two examples demonstrate how to do this. The output activity will send the text typed into its message box if the Live Activity is activated and you click the *Send* button in the browser application. The input activity will put the message into the browser's window if it is activated.

These two web apps do not need to be run together, you can have any combination of all the routable examples. Run two or three of the routable input examples and two of the routable output examples. Try the simple routable output example in Python and the web routable input. Or run the web routable output and the speech routable input. Any message you type into the browser's text box will then be spoken by the speech activity. Pretty cool, eh?

5.3.1 Native Activities

You can start and stop native activities with Interactive Spaces. This makes it easy for you to write activities in openFrameworks and other languages that don't run inside the Interactive Spaces container.

The following examples are found in the *examples/basics/native* folder of the Workbench.

- interactivespaces.example.activity.native

This example uses the Linux *mpg321* to play an audio file found in the activity.

5.4 Comm Examples

There are a variety of examples which allow you to use Interactive Spaces for communication to a variety of hardware devices (through serial and Bluetooth) and external services (such as Twitter and Chat).

5.4.1 Serial Comm

Serial communication lets you communicate with hardware devices that attach via serial ports, often USB in the modern world.

The examples given with Interactive Spaces typically connect to Arduino microcontrollers and read read or write to sensors connected to the device.

These examples are found in the *examples/comm/serial* folder of the workbench.

You need to configure your controller to work with serial, please see the chapter on Interactive Spaces Comm Support for instructions.

- interactivespaces.example.activity.arduino.echo

This Arduino example is very simple. When you activate the Live Activity, it will generate a random 8 bit number and write it to the serial port. The source for this activity includes an Arduino sketch called *Echo* which will read any

bytes which come over the serial connection and write them back. The example will only log the values to keep the example simple, so make sure to look at the controller's logs.

- interactivespaces.example.activity.arduino.analog.trigger

This Arduino example connects to the Arduino and expects a value from an analog port to be written. The Arduino code for the example is included in the workbench.

The activity will write on a route if the value read from the Arduino goes over some value. This gives an example of responding to a hardware event and informing any listening activities of the event. If the speech example is activated, it will speak when the message is sent.

5.4.2 XBee

Interactive Spaces can control XBee radios directly. This makes it possible for you to communicate with remote hardware wirelessly.

The radios must contain the API firmware and be set with AP=2 (escaped protocol).

- interactivespaces.example.comm.xbee.coordinator
- interactivespaces.example.comm.xbee.endpoint

The first example runs on a coordinator radio. When you activate the activity it will first send a local AT AP informational command and log the result. It will then transmit the number 1234 in hex to the endpoint radio. Once the endpoint radio receives the packet, the coordinator activity will print out the status packets. The endpoint activity will log the received packet.

Both activities must be told which serial USB connection the radio is on using the configuration parameter *space.hardware.serial.port*. For example, on a Linux box, a typical value would be */dev/ttyUSB0*.

The coordinator activity needs the configuration parameter *xbee.remote.address64*, which gives the 64 bit address for the endpoint radio. Addresses will look like *0013a200407bd2e3*.

5.5 Hardware

The following examples show howto use various hardware devices. They are found in the *examples/hardware* folder of the Workbench.

5.5.1 Bluetooth Comm

Many wireless devices use Bluetooth for short range wireless communication.

The example with Interactive Spaces lets you use a Wii Remote as part of your space. read the activity documentation to see how to use the example.

- interactivespaces.example.activity.wii.remote

5.6 Misc

The following are a set of examples to show other things you can do with Interactive Spaces. They are found in the *examples/misc* folder of the Workbench.

5.6.1 Topic Bridges

Interactive Spaces makes it possible for Live Activities to communicate with each other. At some point you may find yourself having an event producer and an event consumer which need to talk to each other, but they were not written with each other in mind so their messaging protocols are different.

Topic Bridges make it possible for you to translate from one message protocol to another by writing a tiny script which merely says which field or fields from the source message are combined to create the fields of the destination message.

- interactivespaces.example.activity.bridge.topic

5.6.2 XMPP Chat

Sometimes it would be good if visitors to a space could chat with the space using a chat client.

The example with Interactive Spaces will sign into an XMPP-based chat service, such as Google Chat and echo the chat back to the user chatting with the activity. Instead you could use information that users send to the space to affect the space.

- interactivespaces.example.activity.chat.xmpp

5.6.3 Music Jukebox

Sometimes you would like to use Interactive Spaces to play music or other audio files.

- interactivespaces.example.activity.music.jukebox

The above will play MP3 files. A folder of music is set in the Live Activity's configuration and the example will shuffle play MP3s from this folder when activated.

5.7 Android

Space Controllers can run on Android devices.

The following examples demonstrate writing activities for Android devices and are found in the *examples/android* folder of the Workbench.

- interactivespaces.example.activity.android.simple
- interactivespaces.example.activity.android.web
- interactivespaces.example.activity.android.accelerometer

The first merely logs to the Android logs various Activity lifecycle events. The second will start up a web browser on the phone which opens a window to the Interactive Spaces website. The third will read values from the accelerometer on the Android device and transmit them over a route to any other activities in the space which may be interested.

CHAPTER

SIX

THE WORKBENCH

The Interactive Spaces Workbench provides the ability to create and build projects. You can also create projects for IDEs such as Eclipse.

The Workbench also contains the documentation for Interactive Spaces, which includes both this manual, and the Javadoc for the code implementing Interactive Spaces.

6.1 The Basics

6.1.1 Building an Activity Project

Interactive Spaces projects must be built with the Workbench Builder. From the directory where you installed the Workbench, you can type something like

```
bin/isworkbench.bash <projectdir> build
```

where `<projectdir>` is the folder which contains the project. For example, you can build the Simple Java Activity Example project with the command

```
bin/isworkbench.bash examples/basics/hello/interactivespaces.example.activity.hello build
```

6.1.2 Creating an Activity Project

You can create activity projects very simply.

```
bin/isworkbench.bash create language <language>
```

where `<language>` is one of `java`, `javascript`, `python`, or `android`.

You then get a project of one of those types which contains an initial piece of code for you to then start editing. You will be prompted for identifying name, version, name, and description of the new project.

If you are using `android`, please see the chapter on Android for details about initial configurations.

6.1.3 Using an IDE

if you are doing Java projects, you will want an IDE project. Right now, only Eclipse is supported. You can also use Eclipse for a non-Java project, the same command works.

```
bin/isworkbench.bash <projectdir> ide eclipse
```

where <projectdir> is the directory containing the Activity project.

For example

```
bin/isworkbench.bash foo.bar.yowza ide eclipse
```

will take the foo.bar.yowza project and generate the Eclipse files for it. You can then import the project into Eclipse.

You should rebuild the IDE project every time you add a new JAR into the bootstrap folder of your controller so that you will have access to the classes in that JAR file.

6.2 Project Files

Interactive Spaces Workbench projects use a project.xml file which describes aspects about the project, including its type, its name and description, how to build it, and any dependencies or resources that the project needs.

A basic project.xml file will be created for your project if you create it with the Workbench. There may be a need to extend the contents of this file or to edit it later on, so it is good to understand all of its pieces.

6.2.1 A Basic Activity Project File

Let's look at one of the simplest project files. This is for the interactivespaces.example.activity.hello example which comes with Interactive Spaces.

```
<?xml version="1.0"?>
<project type="activity" builder="java">
  <name>Simple Hello World Activity Example in Java</name>
  <description>
A very simple Java-based activity which just logs at INFO level its
callback messages.
  </description>

  <identifyingName>interactivespaces.example.activity.hello</identifyingName>
  <version>1.0.0</version>
</project>
```

As you can see, project files are XML-based. The root element is called project and has a couple of attributes giving information about the project. One is type which gives the type of the project. Since this example is an activity project, so the type attribute has the value activity.

The Workbench needs to know how to build projects. The build attribute says which builder to use for the project. The interactivespaces.example.activity.hello project is a Java-based project, so uses the Java builder. So we give the attribute builder the value java.

Java projects are somewhat complicated to build because they need to use the Java compiler and create a jar file. Projects that use scripting languages or are web only can use a very simple builder that merely takes the contents of the src/main/resources folder and places them into a properly formatted zip file, the format that Interactive Spaces uses for its activity files. These projects do not need to specify the builder attribute, as you can see in the example project interactivespaces.example.activity.hello.python.

```
<?xml version="1.0"?>
<project type="activity">
  <name>Simple Hello World Activity Example in Python</name>
```

```
  <description>
A very simple Python-based activity which just logs at INFO level its callback messages.
  </description>

  <identifyingName>interactivespaces.example.activity.hello.python</identifyingName>
  <version>1.0.0</version>
</project>
```

Notice that the `project` element only contains the `type` attribute, not the `builder`.

The next part of the project file is the `name` element. This gives the informational name of the project, the name that will appear in the Interactive Spaces Web Admin.

The `description` element gives a more detailed description of the project. It is optional. It is also displayed in the Interactive Spaces Web Admin when looking at the specific page for the Activity.

The `identifyingName` element gives the Identifying Name for the project. This name is used by the internals of Interactive Spaces and has very strict rules on its syntax. The combination of the Identifying Name and the Version uniquely identify the Activity to Interactive Spaces.

The Identifying Name is is a dot separated set of names, Examples would be things like

- a.b.c
- com.google.myactivity

Each part of the name must start with a letter and can then be letters, digits, and underscores.

The `version` element gives the version number of the project. Versions consists of 3 sets of numbers, separated by dots. Examples would be

- 1.0.0
- 0.1.0-beta

Notice the last one has a dash followed by some text.

6.2.2 Resource Copying

Often you will find that you have resources which several of your projects will use. An example would be Javascript libraries that are being used in several of your web browser-based activities.

You can have these resources copied into every project that uses them by putting a `<resources>` section in your `project.xml` file. This section will specify all resources that should be packaged in your project.

An example would be

```
<resources>
  <resource destinationDirectory="webapp/fonts/OpenSans"
      sourceDirectory="${repo}/resources/fonts/OpenSans" />
  <resource destinationDirectory="webapp/js/libs"
      sourceDirectory="${repo}/resources/web/js/base" />
  <resource destinationDirectory="webapp/js/libs"
      sourceFile="${repo}/resources/web/js/external/jquery/core/jquery-1.9.1.min.js" />
</resources>
```

These resource declarations are giving the location of resources that are needed and where they should be copied to.

Lets look at the first one.

```
<resource destinationDirectory="webapp/fonts/OpenSans"
    sourceDirectory="${repo}/resources/fonts/OpenSans" />
```

This gives the destination directory where the resources should be copied to in the `destinationDirectory` attribute. Here the OpenSans font files are being copied to the `webapp/fonts/OpenSans` subfolder of an Activity.

The source of the OpenSans fonts is given by the `sourceDirectory` attribute. The entire content of the source directory will be copied from the source directory, including the content of any subfolders of the source directory, their subfolders, and all the way down.

One thing to notice here is the use of `${repo}`. This is an example of using a local configuration variable to specify where the resources are being copied from or to. The example here is demonstrating having a code repository which contains all resources being used for all projects. See *Configuring the Workbench* for more details on how to declare local configuration variables.

The last entry in the above example shows how to copy a specific file.

```
<resource destinationDirectory="webapp/js/libs"
    sourceFile="${repo}/resources/web/js/external/jquery/core/jquery-1.9.1.min.js" />
```

This `<resource>` element uses the `sourceFile` attribute to specify an exact file to be copied into the destination directory. In this case the file will be copied and given the exact name that the source file has, in this case `jquery-1.9.1.min.js`.

If you want to rename the file, you can use the `destinationFile` attribute instead of the `destinationDirectory` attribute.

```
<resource destinationFile="webapp/js/libs/jquery.js"
    sourceFile="${repo}/resources/web/js/external/jquery/core/jquery-1.9.1.min.js" />
```

The above example would copy the file `jquery-1.9.1.min.js`, but would name it `jquery.js` in the destination location.

Your Project Source Directory

The Workbench supplies a configuration property that gives you the location of your project folder. It is called `project.home` and gives the full filepath to where your project lives. This can be useful for copying resources from your project from, say, a library from a C or C++ build.

An example could be

```
<resource sourceFile="${project.home}/native/build/artifact" />
```

Which would copy the file `artifact` from the subfolder `native/build` in your project folder. `native/build` might be the folder that your C build places the final library or executable that it builds.

Resource Assemblies

In addition to copied resources, it's possible to include an `assembly`, which is a single bundle file that's extracted into a collection of files. See *Assembly Projects* for documentation on how assembly projects are created. To use a resource assembly, specify an `<assembly>` tag as in the example below.

```
<resources>
  <assembly packFormat="zip" sourceFile="${project.home}/javascript.bundle-1.0.0.zip" />
</resources>
```

Additional Sources

Using the `sources` directive in a project, it's possible to include additional directories into the project build source path. For example, this can be used to create a shared source java file that contains constants used across a number of

different activities. The additional sources are passed to the underlying project builder, which will typically process them in the same manner as source files in the activity home directory itself.

Note that is is also possible to use *Library Projects* to create shared activity functionality.

```
<sources>
  <source sourceDirectory="${project.home}/../shared/src/main/java" />
</sources>
```

6.2.3 Quick Importing or Deploying Your Projects

After building a project you will need to import it into the Interactive Spaces Master Web Admin. This can involve a lot of mouse clicks, so Interactive Spaces makes it easy to import or deploy your application from the Workbench command line. For example, something I do a lot is use the command

```
bin/isworkbench.bash my/project/location clean build deploy testdeploy
```

This command would do a clean build of the project contained in the folder `my/project/location` and then deploy it to the `testdeploy` target.

Deploy targets are found in the `project.xml` file in the Deployments section. An example would be

```
<deployments>
  <deployment type="testdeploy" location="${deployment.test.deploy}" />
  <deployment type="testimport" location="${deployment.test.import}" />
</deployments>
```

The command line example given above refers to a deploy target called `testdeploy`. The deployment target is defined with a `<deployment>` element. The `testdeploy` example

```
<deployment type="testdeploy" location="${deployment.test.deploy}" />
```

specifies the deployment target name in the `type` attribute. The Workbench would then copy the activity to the value of the `location` attribute. Here we are using a local configuration variable to specify where the built Activity should be copied to. See *Configuring the Workbench* for more details on how to declare local configuration variables. The value of this variable would be the autoimport folder (see *Importing and Deploying* for details) for the Interactive spaces master for your development installation. You could also provide deployments for your QA environment, your production network, etc.

6.2.4 A Complete Project File

Here is an example of a complete Activity project file with resource and deployment sections.

```
<?xml version="1.0"?>
<project type="activity" builder="java">
  <name>My Web Activity</name>
  <description>
A simple web activity.
  </description>

  <identifyingName>my.web</identifyingName>
  <version>1.0.0</version>

  <resources>
      <resource destinationDirectory="webapp/fonts/OpenSans"
          sourceDirectory="${repo.cec}/resources/fonts/OpenSans" />
      <resource destinationDirectory="webapp/js/libs"
```

```
          sourceDirectory="${repo.cec}/resources/web/js/base" />
      <resource destinationDirectory="webapp/js/libs"
          sourceFile="${repo.cec}/resources/web/js/external/jquery/core/jquery-1.9.1.min.js" />
  </resources>

  <deployments>
      <deployment type="testdeploy" location="${deployment.test.deploy}" />
      <deployment type="testimport" location="${deployment.test.import}" />
  </deployments>
</project>
```

6.3 Other Project Types

6.3.1 Library Projects

Library projects let you write code which can be shared across multiple Interactive Spaces Activities. Libraries are one way in which you can extend the functionality of Interactive Spaces with your own functionality.

An example `project.xml` file for a library project is given below.

```
<?xml version="1.0"?>
<project type="library" >
  <name>Support for Interactive Spaces projects</name>
  <description>
Support For Interactive Spaces projects.
  </description>

  <identifyingName>my.support</identifyingName>
  <version>1.0.0</version>
</project>
```

Library projects must be Java-based, hence the lack of the `builder` attribute on the `<project>` element. The project file has the same name, description, identifying name, and version sections that all projects must have. But the `type` attribute of the `<project>` element has the value `library`.

The artifact built for a Library project will be a Java jar file. It can be copied into the `bootstrap` folder of an Interactive Spaces controller and will then be available for Activities to use.

If you add a new Library into a Controller, make sure you recreate the IDE project for any Activities which will use the Library and refresh the project in your IDE. See *Using an IDE* for more details on creating the IDE project for a Workbench project.

6.3.2 Service Projects

6.3.3 Resource Projects

6.3.4 Assembly Projects

Assembly projects create a bundle file (typically as a compressed `zip` file) that consists of several constituent files. The resulting bundles can then be included using an `assembly` resource directive (see *Resource Assemblies*). The project snippet below will create an assembly named `javascript.bundle-1.0.0.zip` that can then be included elsewhere.

```
<project type="assembly" packFormat="zip" >
  <identifyingName>javascript.bundle</identifyingName>
  <version>1.0.0</version>
  <sources>
    <source sourceFile="src/main/css/base_admin.css" />
    <bundle destinationFile="webapp/js/bundle.js">
      <source sourceFile="src/main/js/external/jquery/core/jquery-1.9.1.min.js"/>
      <source sourceFile="src/main/js/base_admin.js" />
    </bundle>
  </sources>
</project>
```

Internal to the bundle, there is a `base_admin.css` file as well as a bundle file `bundle.js`, which contains two individual source files concatenated together.

6.4 Other Workbench Operations

6.4.1 Configuring the Workbench

You can provide configuration variables to the Workbench which become available during project builds. These configurations would go in a file called `local.conf` and placed in the `config` folder found where you installed your Workbench.

An example of a local configuration file would be

```
repo=/my/home/repo
deployment.test.deploy=/my/home/interactivespaces/master/master/activity/deploy
deployment.test.import=/my/home/interactivespaces/master/master/activity/import
```

This configuration file would make the variables `${repo}`, `${deployment.test.deploy}`, and `${deployment.test.import}` available for your `project.xml` files.

These examples are showing where the code repository being used for the project would be found, useful if there are common resources that you want to use in multiple projects, and also where the Interactive Spaces Master being used for your development work is located. The directories given are folders watched by the master for when files are copied into them which are then automatically imported into the master or deployed to all controllers containing Live Activities based on the Activity being copied.

6.4.2 Creating Documentation for a Project

When creating projects like library projects that may be shared with others, it is important to give good documentation for those who will use the library. The Workbench can create documentation fort your project.

```
bin/isworkbench.bash <projectdir> docs
```

where `<projectdir>` is the folder which contains the project. For example, you can build the Simple Web Activity Example project with the command

```
bin/isworkbench.bash examples/basics/hello/interactivespaces.example.activity.hello docs
```

The Workbench can only create Javadocs at the moment, which means it currently only works on Java-based Activities or Library projects. The output of the command will be placed in the `build/docs` folder of your project.

6.4.3 Performing Workbench Operations on a Collection of Projects

Sometimes you want to build a collection of Interactive Spaces projects. If all of the projects are contained within a given root folder this is easy to do.

```
bin/isworkbench.bash <rootdir> walk <commands>
```

Here `<rootdir>` is the root directory containing all of the projects and `<commands>` is the list of commands to be done on all of the projects.

The `walk` command will walk all subfolders of the root directory looking for folders which contain a `project.xml` file. Those it finds it will perform the commands on.

For instance, if you want to do a clean build all of the examples which come with the Workbench, you could use the command

```
bin/isworkbench.bash /my/home/interactivespaces/workbench/examples walk clean build
```

where `/my/home/interactivespaces/workbench` would be the directory where you installed the Workbench.

6.4.4 Adding Flags to the Java Compiler

You can add additional flags to the Java compiler by defining the local configuration variable `interactivespaces.workbench.builder.java.compileflags` in your local Workbench configuration. This will add the space-separated flags to the command line to the Java compiler.

See *Configuring the Workbench* for more details on how to declare local configuration variables.

6.4.5 OSGi Bundle Wrapping

Interactive Spaces uses a Java technology called OSGi as its runtime container. OSGi permits Interactive Spaces to do many things, such as run multiple versions of the same library or Live Activity, even if the different versions are binary incompatible with each other.

For this to work, the libraries that Interactive Spaces uses must be made into what is called an OSGi bundle. Many open source libraries are already OSGi compatible, but not all are. Because of this, the Interactive Spaces Workbench provides a way of making a Java jar into an OSGi compatible one.

To create an OSGi bundle, you use the command

```
bin/isworkbench osgi <pathToJar>
```

where `<pathToJar>` is the path to the Java library you want made into an OSGi bundle. The output of the command will be a new jar with the prefix `interactivespaces.` added to its name. For instance, if your jar was originally called `foo-1.0.0.jar`, the OSGi bundle created from the jar will be called `interactivespaces.foo-1.0.0.jar`.

6.5 Best Practices for Developing in Interactive Spaces

There are some simple things you can do if you want to develop reasonably quickly in Interactive Spaces.

6.5.1 Importing and Deploying

It is important to understand the difference between importing and deploying in Interactive Spaces. Importing an Activity places an Activity into the Master's Activity Repository and makes it possible to find from the Master Web Admin. It is normally done from the **Activity** tab in the Master Web Admin.

However importing an Activity into the Master does not immediately send it to the Space Controllers where instances of the Activity are deployed. To do that you need to deploy the Live Activities which are based on the Activity you are developing.

You should create the following two folders where your Master is installed

- `/my/home/interactivespaces/master/master/activity/import`
- `/my/home/interactivespaces/master/master/activity/deploy`

where `/my/home/interactivespaces/master` is where your Master is installed.

Yes, that bit in the middle is real, it is not a stutter, it really is meant to be `master/master`. These two folders are watched by the master and are used for automatically importing or dpeloying Activities or the Live Activities they are based on.

If you copy an Activity into the `import` folder, it is the same as importing it from the Master Web Admin. The Activity will be created in the Master if no other activity has the same Identifying Name and Version. If an Activity has the same Identifying Name and Version the Activity just imported will replace the old Activity that was in the Activity Repository.

If you copy the Activity into the `deploy` folder, it will first be imported into the Activity Repository using the same rules given above. Then any Live Activities based on the Activity will be re-deployed to the Space Controllers they are on. If you use the Deployment project file section discussed in *Quick Importing or Deploying Your Projects*. This means you can compile and deploy to the controller in one fell swoop.

CHAPTER

SEVEN

ACTIVITIES

Activities are the workhorse of your Interactive Space. Any functionality that you want is implemented as an Activity.

Activities can be as simple as a few lines in a configuration file or as complicated as a full set of Java classes. Don't worry if you don't know Java, you can also write Live Activities in a variety of scripting languages. *Activity Types* allow you to chose what kind of Activity you will implement.

Basic Activity Functionality is what functionality you will find, no matter what kind of Activity is implemented.

Supported Activity Classes let you easily write Activities by just supplying a new methods that supply the needed functionality.

Activity Configurations let you take Activities and configure them for their exact use in a particular instance.

Basic Activity Communications allow Activities to communicate events and other information to each other.

Activity Components let you easily add additional functionality to Supported Activity Classes.

There is a lot of information in the next few chapters because there is a lot to Activities.

CHAPTER

EIGHT

ACTIVITY TYPES

Activity Types make it easy for you to implement an Activity with mostly configuration parameters and only as much code as is necessary to add in the behavior you want.

8.1 Web Activity

The simplest Live Activity at all is a *Web* Activity. Web Activities will automatically start up a web browser and web server when the Live Activity starts up. The web server will have content supplied with the Activity and the browser will start up pointing at that content.

8.1.1 Web Activity Configuration

The configuration for a web Activity is pretty simple, the simplest `activity.conf` is given below.

```
space.activity.type=web
space.activity.name=webExample

space.activity.webapp.content.location=webapp
space.activity.webapp.url.initial=index.html
```

The configuration parameter `space.activity.type` gives the Activity Type, which here is `web`.

The configuration parameter `space.activity.webapp.content.location` gives the location of the content to be served. Here the folder it will use (which will contain the HTML, CSS, images, Javascript, and other web resources) is relative to to where your Live Activity is deployed. In the example above, this is a subfolder called `webapp`.

The configuration parameter `space.activity.webapp.url.initial` gives the initial webpage that will be shown in the browser. In the example here, the initial URL is `index.html`, so the browser URL will be `http://localhost:9000/webExample/index.html`.

Notice that `webExample` is part of the URL. The name of the Activity is used as the initial part of the URL. The Activity name is set with the required configuration parameter `space.activity.name`.

The file `index.html` will be found in the location

```
live activity install dir/webapp/index.html
```

where `live activity install dir` is where the Live Activity is installed on the Space Controller.

You may notice that the web server port is `9000`. This is the default server port for the web server. If you want to set it to another value, say for example you need to run multiple web servers on the same machine, or

port 9000 is being used for something else on the computer, you can change it with the configuration parameter `space.activity.webapp.web.server.port`

For example, if you have

```
space.activity.webapp.web.server.port=9091
```

in your `activity.conf`, the initial URL will be `http://localhost:9091/webExample/index.html`.

Sometimes there may be a need for some query string parameters on that initial URL. A query string parameter list can be created using the configuration parameter `space.activity.webapp.url.query_string`. For example, the setting

```
space.activity.webapp.url.query_string=foo=bar
```

would make the initial URL be `http://localhost:9091/webExample/index.html?foo=bar`.

The query string parameters are best used when created from combinations of other configuration parameters. for instance, if your `activity.conf` included the following parameter

```
space.activity.webapp.url.query_string=foo=${a}&bar=${c}
a=b
c=d
```

the initial URL would be `http://localhost:9091/webExample/index.html?foo=a&bar=b`.

The final configuration parameter says whether or not the browser should be in debug mode or not. The default value is to not be in debug mode and the browser will be started in kiosk mode if that is possible. In debug mode, the browser will be opened as normal so that the browser's debugger can be used to debug your Activity.

The configuration parameter which sets whether or not the browser is started in debug mode is `space.activity.webapp.browser.debug`. It should be given a value of `true` or `false`. An example would be

```
space.activity.webapp.browser.debug=true
```

8.1.2 Multiple Browsers

Interactive Spaces starts every browser instance with its own profile. This means that you can start up the browser-based Live Activities on the same machine where you have your normal browser open and they won't affect each other.

8.2 Native Activity

Native Activities give you the ability to run native programs on your computer. Native programs could be ones that came with the operating system the computer runs, or a C++ activity that you write in a framework like openFrameworks. Pretty much it can be any program that you can start from the command line of your operating system's shell.

8.2.1 Native Activity Configuration

A pretty simple `activity.conf` for a Native Activity is given below.

```
space.activity.type=native
space.activity.name=nativeExample

space.activity.executable.linux=my_mp3_player
space.activity.executable.flags.linux=-q ${activity.installdir}/NativeActivityExample.mp3
```

Here you can see that the Activity Type is `native`. Notice there is also the other required configuration parameter `space.activity.name`.

The next two lines give the executable to run and any command line flags that the executable might need.

The first one gives the executable with the configuraton parameter `space.activity.executable.linux`. Here it has the value `my_mp3_player`. If you were running your Live Activity on a Linux box, Interactive Spaces would start up the program

```
live activity install dir/my_mp3_player
```

where `live activity install dir` is where the Live Activity is installed on the Space Controller.

Notice the `.linux` on the end of the configuration parameter name. This specifies which operating system this particular executable is for. his way you can create a Universal Activity which contains executables for any operating system the Activity might run on. Legal values for the moment are

- linux - A Linux computer
- osx - A Mac OSX computer
- windows - a Windows conputer

As an example, the `activity.conf` might contain

```
space.activity.executable.linux=my_linux_mp3_player
space.activity.executable.osx=my_osx_mp3_player
space.activity.executable.windows=my_windows_mp3_player
```

This would mean the Activity would contain the 3 executables

- my_linux_mp3_player
- my_osx_mp3_player
- my_windows_mp3_player

and Interactive Spaces will pick the correct executable based on the OS the Activity is running on.

Often there may be a need for command line arguments, for instance, the mp3 player needs to know which song to play. In the example above, the configuration parameter `space.activity.executable.flags.linux` gives the command line flags when the Linux executable is being used.

The value you see

```
space.activity.executable.flags.linux=-q ${activity.installdir}/NativeActivityExample.mp3
```

gives the command line flags to play a file which is in the Live Activity's install directory on its Space Controller.

```
live activity install dir/NativeActivityExample.mp3
```

where `live activity install dir` is where the Live Activity is installed on the Space Controller.

The executable can also be somewhere else on the machine the Activity is running on. For example, the `activity.conf` below uses the program `/usr/bin/mpg321` to play the MP3 file that comes with the Activity.

```
space.activity.type=NATIVE
space.activity.name=nativeExample

space.activity.executable.linux=/usr/bin/mpg321
space.activity.executable.flags.linux=-q ${activity.installdir}/NativeActivityExample.mp3
```

8.2.2 Native Activities Automatic Keep Alive

Every once in a while, a native activity may crash. Interactive Spaces tries to keep things alive, and this is particularly true for native activities. If, for instance, you shut the web browser down or otherwise kill it, you will notice it starts up again for some limited number of times.

8.3 Scripted Activity

A lot of people feel uncomfortable programming in Java. Programming in Java gives the most direct access to the power of Interactive Spaces, but Scripted Activities do have a lot of advantages. You can edit them directly from their installation folder, which helps a lot when you are writing your Activity in the first place.

Interactive Spaces supports writing Activities in Javascript and Python, with more languages coming soon.

8.3.1 Scripted Activity Configuration

A simple `activity.conf` for a Scripted Activity is given below.

```
space.activity.type=script
space.activity.name=activityPythonScriptExample

space.activity.executable=ExamplePythonActivity.py
```

Notice that the Activity Type is `script`.

The important configuration parameter here is `space.activity.executable` which gives the Activity executable. Here it has the value `ExamplePythonActivity.py`. Interactive Spaces uses the file extension to determine the scripting language being used.

The guaranteed extensions are

Language	Extensions
Javascript	js
Python	py

8.3.2 Scripting Paths

Scripted Activities can use more than 1 scripting file for their implementation. Interactive Spaces supports 2 places for scripting libraries to be placed, one at the Space Controller-wide level, and one at the per-Live Activity level.

Container Wide Paths

The Space Controller-wide scripting library path is in the `interactivespaces/controller/lib` folder. For example, `interactivespaces/controller/lib/python` contains the Python libraries which can be used by every Python script in Interactive Spaces.

`interactivespaces/controller/lib/python/PyLib` contains the Python system libraries.

`interactivespaces/controller/lib/python/site` is where you should put any of the libraries you want to include. Every directory in the `site` directory is automatically added to the Python path.

per-Activity Paths

Any files found in the subdirectory `lib/python` in the Live Activity's install folder will also be added to the Python path. For example, suppose the UUID of your Live Activity (which you can find on the Live Activity's page in the Interactive Spaces Master webapp) is `34eb3c27-5d37-45aa-a9cd-22d46bc85701`. The per-Live Activity Python lib path for that specific Live Activity would then be found in the folder

```
interactivespaces/controller/controller/activities/installed/
    34eb3c27-5d37-45aa-a9cd-22d46bc85701/install/lib/python
```

8.4 Interactive Spaces Native Activities

Interactive Spaces Native Activities (not to be confused with Native Activities) are Activities written in Java that have direct access to all of Interactive spaces services. This is true of some of the scripting languages as well, but Interactive Spaces Native Activities guarantee access to everything.

8.4.1 Interactive Spaces Native Activity Configuration

A simple `activity.conf` for a Interactive Spaces Native Activity is given below.

```
space.activity.type=interactivespaces_native
space.activity.name=example_activity_java_simple

space.activity.executable=interactivespaces.example.activity.java.simple-1.0.0.jar
space.activity.java.class=interactivespaces.activity.example.java.simple.SimpleJavaExampleActivity
```

Notice that the Activity Type is `interactivespaces_native`.

The executable this time is a Java jar file which contains all of the classes needed by the Activity. The Workbench IDE builds this jar file for you with all of the things that it needs, like the OSGi manifest headers.

The important configuration parameter is `space.activity.java.class`, which gives the name of the Java class which is the Activity. here it has the value `interactivespaces.activity.example.java.simple.SimpleJavaExampleActivity`.

CHAPTER

NINE

BASIC ACTIVITY FUNCTIONALITY

There is a basic set of functionality which is accessible from any Activity no matter how it is implemented. This includes access to logging, the state of the Activity, and the Space Controller that the Activity is running in.

If you are using *Supported Activities* you can just use these methods. If you are implementing your own Activities completely from scratch, which you should rarely have to do, you will have to implement all of them yourself.

The definitive documentation for activities is found in the Activity Javadoc.

9.1 General Activity Information

There are several methods in Activities which give information about the Activity.

`getName()` return the name of the Activity as set in the `activity.conf`.

`getUuid()` returns the UUID of the Activity when it was added to a Live Activity. The UUID is assigned by Interactive Spaces when an a Live Activity is created. The Master uses the UUID to control the Live Activity through the Space Controller.

9.2 The Activity Configuration

The Activity's configuration can be accessed with the `getConfiguration()` method.

You can find out more details of an Activity's configuration in the *Activity Configurations* chapter and in the Configuration Javadoc.

9.3 Activity Status

There are several method which can be called to get or set the status of an Activity.

9.3.1 Getting the Activity Status

An Activity can check its own status by calling `getActivityStatus()`. This returns an ActivityStatus object (ActivityStatus Javadoc) which contains information about the current status of the Activity.

9.3.2 Is the Activity Activated?

Activities have a method which can say whether or not the Activity has been activated.

`isActivated()` returns `true` if the Activity is activated and `false` otherwise.

9.3.3 Setting the Activity Status

An Activity can change its status by calling `setActivityStatus()`. This method takes an ActivityStatus object (ActivityStatus Javadoc) as its only argument.

There should be little reason for you to use this method if you are using *Supported Activities*.

9.4 Logging

There are per-Live Activity logs if you want to look at information specific to only one particular Live Activity.

This log is accessed via the `getLog()` method of your Activity.

The per-Live activity logs are found in the Live Activity's folder on the Space Controller.

Suppose the UUID of your Live Activity (which you can find on the Live Activity's page in the Interactive Spaces Master webapp) is *34eb3c27-5d37-45aa-a9cd-22d46bc85701*. The logs for that specific Live Activity would then be found in the folder

```
interactivespaces/controller/controller/activities/installed/
    34eb3c27-5d37-45aa-a9cd-22d46bc85701/log
```

The configuration parameter *space.activity.log.level* lets you set the logging level, which by default is set to *error*. The following parameter added to your *activity.conf* will take the logging level up to *info*.

```
space.activity.log.level=info
```

Legal values are

- *fatal* - Fatal events.
- *error* - Error and fatal events
- *info* - Info, error, and fatal events
- *debug* - Debug, info, error and fatal events
- *off* - No logging

9.5 The Activity Filesystem

When a Live Activity is installed on a Space Controller, it is placed in a directory with a number of subdirectories which contain a variety of files needed for the Live Activity.

The Activity Filesystem is accessed with the `getActivityFilesystem()` call.

A Live Activities is installed in the

```
controller/controller/activities/installed/uuid
```

folder, where `uuid` is the UUID of the Live Activity.

Each of the following directories are under this directory.

9.5.1 The Install Directory

The Install Directory is where the resources that were contained in the Activity's install bundle are placed. This includes the `activity.conf` and any code and resources necessary for the Activity to run.

The Install Directory is in the `install` directory of the Activity's fileystem.

This directory is accessed with the `getInstallDirectory()` method on the Activity Filesystem.

```
File installDir = getActivityFilesystem().getInstallDirectory();
```

You can access a specific file in the install directory with the `getInstallFile()` method. Say, for example, you want to access the file `data.dat` which is in the `resource` subdirectory of the install directory.

```
File dataFile = getActivityFilesystem().getInstallFile("resource/data.dat");
```

9.5.2 The Permanent Data Directory

The Permanent Data Directory is where a Live Activity can place data it wishes to keep around permanently. Interactive Spaces guarantees it will not touch this folder unless the Live Activity is deleted from the controller.

The Permanent Data Directory is in the `data` directory of the Activity's fileystem.

This directory is accessed with the `getPermanentDataDirectory()` method on the Activity Filesystem.

```
File dataDir = getActivityFilesystem().getPermanentDataDirectory();
```

You can access a specific file in the permanent data directory with the `getPermanentDataFile()` method. Say, for example, you want to access the file `data.cache` which is in the `cache` subdirectory of the permanent data directory.

```
File dataFile = getActivityFilesystem().getPermanentDataFile("cache/data.cache");
```

9.5.3 The Temporary Data Directory

The Temporary Data Directory is where a Live Activity can place data it wishes to keep around while it is running. Interactive Spaces only guarantees that it won't delete this directory while a Live Activity is running. Any data which needs to be kept between runs should be put in the *The Permanent Data Directory*.

The Temporary Data Directory is in the `tmp` directory of the Activity's fileystem.

This directory is accessed with the `getTempDataDirectory()` method on the Activity Filesystem.

```
File tmpDir = getActivityFilesystem().getTempDataDirectory();
```

You can access a specific file in the Temporary Data Directory with the `getTempDataFile()` method. Say, for example, you want to access the file `data.cache` which is in the `cache` subdirectory of the Temporary Data Directory.

```
File cacheFile = getActivityFilesystem().getTempDataFile("cache/data.cache");
```

9.5.4 The Log Directory

The Log Directory is where a Live Activity places its *per-Activity logs*.

The Log Directory is in the `log` directory of the Activity's fileystem.

This directory is accessed with the `getLogDirectory()` method on the Activity Filesystem.

```
File logDir = getActivityFilesystem().getTLogDirectory();
```

9.6 The Space Controller

The Activity can access the Space Controller which it is running under.

The Space Controller is accessed with the `getController()` call.

See the SpaceController Javadoc for details.

9.7 The Space Environment

The Space Environment gives access to many of the core Interactive Spaces services. You can find more information in the chapter *The Space Environment*.

The Space Environment is accessed through the `getSpaceEnvironment()` call in an Activity.

CHAPTER

TEN

SUPPORTED ACTIVITY CLASSES

Implementing an Interactive Spaces Activity from scratch, even with the vast number of support classes available, can be daunting. To help with this, Interactive Spaces comes with a variety of Supported Activities which can make implementing an Activity as simple as writing methods for only the functionality that you care about.

10.1 Common Functionality

There is a set of functionality which is common to all of the Supported Activity classes.

10.1.1 Common Event Methods

There are various events which happen to an Activity, they can be started up, shutdown, activated and deactivated. There are a series of event methods which are called during these various Activity events that you can write your own versions of so that your Activity does what you need it to do. You don't have to write versions of all of these methods, but you will probably need to write some of them.

Sometimes multiple event methods will be called for a single event so that the Activity can respond properly to its internal state, so make sure you carefully read when event methods are called.

`void onActivitySetup()`

This method is called when Activity is starting up. This is the first event method called and is called only once.

> This event is usually used to register Activity Components.

Very minimal things will be set up for the Activity when this event happens. The method should be used to set anything up that doesn't require access to Activity Components or any other configured items. This means that routes, web socket servers, and other communication channels are not available.

The following resources are available:

- the *space environment*
- the Activity's initial configuration
- the Activity's log
- The Space Controller the Activity is running in

This method should throw an exception if it can't properly setup. This will cause the Activity startup to fail.

void onActivityStartup()

This method is called once when the Activity is starting up. It is called after the Activity is fully configured and all Activity components, such as web servers and communication channels, are running, though are not processing events yet.

This event is typically used to set up any data structures or other non-Activity OCmponent-based resources you will use, like Managed Commands.

Incoming messages on routes and web socket communications with the web servers are not handled until this method has completed, though any messages which come in are queued for later processing.

It is not recommended that messages be sent out during this method, there is no guarantee that any of the communication channels are fully ready. Messages should be sent in ref:*onActivityPostStartup <onActivityPostStartup-reference-label>*.

This method should throw an exception if it can't properly start. This will cause the Activity startup to fail.

It is called after *onActivitySetup*.

void onActivityPostStartup()

This method is called once after the Activity has completed starting up.

At this point everything is fully running, event handlers are processing events, and it is completely safe to start sending messages via routes.

Do be aware, though, that not all web socket connections will necessarily be made at this point. Initial data for web socket can be created in the *onActivityStartup* event and sent once the web socket `onConnect` event is received.

If this method throws an exception, the Activity will still start.

It is called after *onActivityStartup*.

void onActivityPreShutdown()

This method is called first when the Activity is shutting down. It provides an opportunity to send out messages on any communication channels before those channels are shutdown in *onActivityShutdown*.

void onActivityShutdown()

This method is called when the Activity is shutting down. It should be used to properly shut down anything that the Activity needed that wasn't automatically supported (such as components). Any communication channels such as routes or websockets will not be available during this call.

This method should throw an exception if it can't shutdown.

This method is called after *onActivityPreShutdown*.

Do consider doing any shutdown cleanup of your Activity in *onActivityCleanup* as it is called whether the Activity shuts down or crashes.

void onActivityActivate()

The Activity is being activated.

This method should throw an exception if the Activity can't activate.

void onActivityDeactivate()

The Activity is being deactivated.

This method should throw an exception if the Activity can't deactivate.

void onActivityFailure()

Something in the Activity has failed. This can be any installed components or something the user has set up.

void onActivityCleanup()

The Activity has shut down either due to a shutdown or by activity failure. It should clean up all resources used by the Activity.

It is called after *onActivityShutdown* is called during shutdown, or when the Activity crashes.

boolean onActivityCheckState()

This method will be called when the activity state is being checked by the controller.

This method should not change the activity state, it should just return whether or not the activity is doing what it is supposed to in its current state.

The method should return `true` if the Activity is working correctly, and `false` if it isn't.

void onActivityConfigurationUpdate(Map<String, Object> update)

A live configuration update is coming into the Activity.

The map gives the contents of the entire update.

The new configuration will also be properly reflected with the `getConfiguration()` method on the Activity.

10.1.2 Thread Pools and Managed Commands

It is sometimes necessary to run several things at the same time in your Activities and the typical way to do that is with threads. However, threading in Interactive Spaces can be a little tricky because you want the Master or Space Controller to shut down when you want it shut down. If threads are not used properly, your Master or Space Controller will not shut down because there are threads still running.

Managed Commands give you a per-Activity collection of threads which will all be properly shut down when your Activity is cleaned up.

You can access the Managed Commands with the `getManagedCommands()` call in your Activity.

To use the Managed Commands service, you can create a `Runnable` inside your *onActivitySetup* or *onActivityStartup* and submit it to the Managed Commands.

```
public void onActivitySetup() {
  ... other setup...

  Runnable myTask = ...
  getManagedCommands().submit(myTask);
```

```
  ... other setup ...
}
```

You are now done, you don't have to worry about shutting your task down, Interactive Spaces will do it automatically when the Activity is cleaned up.

For more details, see the CommandCollection Javadoc.

10.1.3 Managed Resources

Some of the provided Interactive Spaces functionality needs to be started up and shut down because of how it works on the inside. As an example, there is a support class for easily copying resources needed for your Activity from an arbitrary URL. This complex support class can work in the background and thus needs to be shutdown when it is no longer used. Because it is a Managed Resource, however, you don't need to remember to start it up or shut it down, it will be taken care of for you automatically.

You tell your Activity about a Managed Resource with the `addManagedResource()` call.

```
public void onActivitySetup() {
  ... other setup...

  httpCopier = new HttpClientHttpContentCopier()
  addManagedResource(httpCopier);

  ... other setup ...
}
```

You are now done, you don't have to worry about shutting the copier down, Interactive Spaces will do it automatically when the Activity is cleaned up.

CHAPTER

ELEVEN

ACTIVITY CONFIGURATIONS

One of the more powerful parts of Interactive Spaces is the Activity Configuration.

11.1 Common Configuration Parameters

There are some values found in every activity configuration which you might find useful.

activity.installdir

This is the directory where the live activity was installed. It will contain the base activity configuration, the activity descriptor, and all other resources which were part of the Activity.

activity.logdir

This is the directory which stores the live activity's log files.

activity.datadir

This is a private directory where the live activity can store data which needs to persist for some time. This data will only be deleted if the live activity is deployed again.

activity.tmpdir

This is a private directory where the live activity can store temporary data. This data can be deleted whenever the live activity is not running.

system.datadir

This folder is writable by any live activity. Any files here will persist between controller shutdowns and startups, and live activity deployments.

system.tmpdir

This folder is writable by any live activity. Files here can be deleted during controller startup or after controller shutdown.

CHAPTER

TWELVE

BASIC INTERACTIVE SPACES COMMUNICATIONS

Bringing Live Activities up and down is nice and all, but if you really want interesting behaviors in your space, you need your Live Activities to communicate with each other.

Interactive Spaces uses ROS to provide communication. Using ROS directly gives the most power, but it is somewhat complex to use. So Interactive Spaces provides a simpler-to-use mechanism called *routes* based on the popular communication format JSON which is used in web applications for transferring name/value pairs between Live Activities. Routes use ROS under the covers, but other than a few configuration parameters that are ROS specific, you need never think about ROS.

Direct use of ROS will be covered in another chapter.

12.1 Route Basics

The message format for route communication is very simple, a collection of name/value pairs. The collection is anything serialization as a JSON message, in fact the message is transmitted on the network as a JSON-encoded string.

The basic Interactive Spaces communication system uses ROS for Activity to Activity and Master to Controller communication. ROS uses the concept of Global Topics, which you can think of as a globally defined mailbox that anyone can write to and anyone can read from, as long as they have the name of the topic. `/example/routable/channel1` is the global ROS Topic that many of the example routable Activities in the Workbench can write to or read from. Every Activity that wants to use the route to communicate must use the same name for the global topic.

Routes can have multiple activities writing information to the route and multiple activities reading from that route. A given activity can write to the channel and never read from it and visa versa. In the examples found in the Workbench you can see one example which only writes on the route and another which only reads from the route.

Routes are used in Activities by giving them a name which is local to the Activity. If the Activity wants to write to a route, it uses this local name, also called a Channel. This name is different than the global topic name.

A lot of words... what does it all mean???

12.1.1 Configuring Routes

The example Workbench Activity `interactivespaces.example.activity.routable.input` reads from a route whose global topic name is `/example/routable/channel1`. The Activity has the following lines in its `activity.conf`:

```
space.activity.routes.inputs=input1
space.activity.route.input.input1=/example/routable/channel1
```

The example Workbench Activity `interactivespaces.example.activity.routable.output` writes to the same route. The Activity has the following lines in its `activity.conf`:

```
space.activity.routes.outputs=output1
space.activity.route.input.output1=/example/routable/channel1
```

Notice that the configuration property `space.activity.route.input.input1` has the same value as the configuration property `space.activity.route.output.output1`. This means that writing to channel `output1` in Activity `interactivespaces.example.activity.routable.output` will show up on channel `input1` in Activity `interactivespaces.example.activity.routable.input`.

`input1` and `output1` are examples of the local name part of a route. These names, once again, are local to an Activity, and can be anything the activity wants it to be. Even names like `saxophone`, `foo`, and `television` would be fine. Of course you should pick names that actually mean something for what the route is used for.

The property `space.activity.routes.inputs` would contain the local names of all route channels the activity will read from. Every channel to be read from will be listed in this property, with each name separated by a `:`. For example

```
space.activity.routes.inputs=foo:bar:bletch
```

would create input channels named `foo`, `bar`, and `bletch`.

For each named input channel, there must be a corresponding property whose name starts with `space.activity.route.input`. For example, in the example above, we have an input route named `input1`, so we must have a property with the name `space.activity.route.input.input1`. The value of this property would be the name of the global topic for the route.

Multiple global topics can be listed as the value for the `space.activity.route.input` property, once again separated by a `:`. This means the channel will listen on all topics listed at the same time.

Similarly

```
space.activity.routes.outputs=foo:bar:bletch
```

would create output channels named `foo`, `bar`, and `bletch`.

For each named output channel, there must be a corresponding property whose name starts with `space.activity.route.output`. For example, in the example above, we have an output route named `output1`, so we must have a property with the name `space.activity.route.output.output1`. The value of this property would be the name of the global topic for the route.

Multiple global topics can be listed as the value for the `space.activity.route.output` property, once again separated by a `:`. This means the channel will write to all topics listed at the same time.

Any Activity which uses ROS communication, remember that routes are implemented using ROS communication, must have the `space.activity.ros.node.name` configuration property defined. This name should be unique for your space, and one way to do that is to make it a relative name, meaning don't start it with a `/`. Names which start with a `/` are absolute names, and should only be used if you know what you are doing and have a good reason for it.

12.2 Using Routes In Code

12.2.1 Routes with `BaseRoutableRosActivity`

The simplest way to use a route is to base your Activity on the `BaseRoutableRosActivity` Supported Activity class.

To read from the route, implement the `onNewInputJson` method. This method has two arguments, one which gives the local name of the channel which received the message, and the second which gives the map of name/value pairs from the message.

This method will be called for any incoming route messages, regardless of which route it came from. Use the first argument to decide which route the message came from.

```
public class SimpleJavaRoutableInputActivity  extends BaseRoutableRosActivity {

    @Override
    public void onNewInputJson(String channelName, Map<String, Object> message) {
        getLog().info("Got message on input channel " + channelName);
        getLog().info(message);
    }
}
```

To write to a route, create a map of name/value pairs and call the `sendOutputJson` method. The first argument will be the name of the output channel you want to write to, the second argument will be the map of name/value pairs to send.

```
public class SimpleJavaRoutableOutputActivity extends BaseRoutableRosActivity {

    @Override
    public void onActivityActivate() {
        Map<String, Object> message = Maps.newHashMap();
        message.put("message", "yipee! activated!");
        sendOutputJson("output1", message);
    }

    @Override
    public void onActivityDeactivate() {
        Map<String, Object> message = Maps.newHashMap();
        message.put("message", "bummer! deactivated!");
        sendOutputJson("output1", message);
    }
}
```

12.2.2 An Advanced Example of Using Routes

It would be good to look at two example projects in the Workbench which demonstrate a very common setup, a web browser Activity which is used to control a remote Activity.

Look at the following two Activity examples in the workbench:

1. examples/basics/comm/interactivespaces.example.activity.routable.output.web
2. examples/basics/comm/interactivespaces.example.activity.routable.input.speech

The first example is a browser-based activity which will start up a web server which serves a webapp and starts up a web browser which displays the webapp to the user. This example shows how to create a web page which can

communicate or obtain information in a browser-based interface. The browser speaks to the IS side via a web socket connection.

The second example creates an instance of a speech synthesizer that can speak text supplied to it.

The two activities talk to each other over a route.

The communication flow is as follows:

1. The user types in text to be spoken into a textbox in the web browser.
2. The user clicks the send button in the browser.
3. The contents of the text box is sent over a websocket connection to Activity 1.
4. Activity 1 packages up the information sent from the browser and sends it over a route to Activity 2.
5. Activity 2 receives the route message sent from Activity 1 and gives it to the speech synthesizer.

CHAPTER

THIRTEEN

ADDITIONAL ACTIVITY FUNCTIONALITY

CHAPTER

FOURTEEN

ACTIVITY COMPONENTS

Activity Components make it easy to add functionality to your Activity. There are components for web browsers and web servers, running native applications, providing communication between activities, and many other things.

The components are very much tied into the lifecycle of an Activity. They are configured from the Activity's configuration. They are automatically startup up and shut down with the Activity so that you don't need to worry about it. Their status is automatically sampled by the Activity.

Some components depend on other components being available. Interactive Spaces makes sure that the components are configured and started up in their dependency order. They are shut down in reverse dependency order.

If a component needed by a requested component isn't available, it will be automatically installed. Because off this, if you find your Activity doesn't start and there is something about some missing configurations that you didn't think you needed, it might be because a missing but needed component was automatically added.

14.1 web.browser

The `web.browser` component will start a web browser on your controller. This browser can be given any URL to start with and can be either a URL external to your Activity or one found within your Activity itself.

14.2 web.server

The `web.server` component starts up a web server within your Activity. This web server can both serve content found in your Activity and also provide a web socket connection for the Activity or anything else within your Interactive Space which needs web socket support.

14.3 native.runner

The `native.runner` component allows your Activity to start and stop applications native to the Operating System on the computer running a Space Controller. This can include music programs, OpenFrameworks applications written in languages like C++, or any other native application.

The configuration parameter

`space.activity.component.native.executable`

gives the path to the executable to be run. It can be relative to the install directory for your Activity on the Controller or can be outside your Interactive Spaces installation. Code outside your installation have to be cleared by a whitelist of allowed applications.

The full name this parameter will have the operating system the controller is running on added to the end. For example, if your Controller is on a Linux computer, the full name for the configuration property is

```
space.activity.component.native.executable.linux
```

This allows you to have multiple versions of an executable in your Activity and the Controller can pick which one to use based on the operating system the Activity is run on.

The configuration parameter

```
space.activity.component.native.executable.flags
```

gives any flags that need to be given to the executable when it runs. As with the executable path, the Controller will add the name of the operating system to the end of this configuration parameter before looking it up in the Activity configuration. For example, if the Operating System for the controller was a Mac running OS X, the parameters will name would be

```
space.activity.component.native.executable.flags.osx
```

CHAPTER

FIFTEEN

THE SPACE ENVIRONMENT

The Space Environment is the access point for many of the core services needed in Interactive Spaces. It is available to Live Activities, Space Controllers, and the Master.

The definitive documentation is found in the InteractiveSpacesEnvironment Javadoc.

15.1 Logging

The Space Container gives access to logging at the container level. There you can find everything logged that happened in both the Master and in the Space Controller. This includes any logging done in the Live Activities.

`spaceEnvironment.getLog()` gives access to this log.

If the root folder for your master is *interactivespaces*, then the container logs are found in the folder

```
interactivespaces/master/logs
```

If the root folder for your controller is *interactivespaces*, then the container logs are found in the folder

```
interactivespaces/controller/logs
```

Activity logging should be done with the usual *activity logging*.

15.2 Getting Services

Services provide more advanced functionality to your Activities. Services provide the ability to do such things as run scripts, schedule future events, send email, and receive email.

Services are obtained through the Service Registry, which is available from the `getServiceRegistry()` call on the Space Environment.

If your Live Activity is written in Java, you can get the Service Registry with the following code

```
getSpaceEnvironment().getServiceRegistry()
```

Once you have the Service Registry, you use the *getService()* method to get an actual service. For instance, if you want the scripting service you would use the following Java code

```
ScriptService scriptService =
    getSpaceEnvironment().getServiceRegistry().getService("scripting");
```

A safer way to do it would be

```
ScriptService scriptService =
    getSpaceEnvironment().getServiceRegistry().getService(ScriptService.SERVICE_NAME);
```

You can get more details about the Service Registry in the ServiceRegistry Javadoc.

You can read more about services in the *Services* Chapter.

15.2.1 Thread Pools

It is sometimes necessary to run several things at the same time and the typical way to do that is with threads. However, threading in Interactive Spaces can be a little tricky because you want the Master or Space Controller to shut down when you want it shut down. If threads are not used properly, your Master or Space Controller will not shut down because there are threads still running.

Interactive Spaces maintains a thread pool which it maintains control over. Al threads from this pool will be properly shut down when Interactive Spaces shuts down and only threads from this thread pool should be used. You should never create a thread on your own.

You can access the thread pool through the `spaceEnvironment.getExecutorService()` call.

If you need threads in your Activity and are using Supported Activities, you should make use of the *Managed Commands* functionality which gives a per-Activity thread pool which will be properly shut down when the Activity is cleaned up.

CHAPTER

SIXTEEN

SERVICES

Services provide more advanced functionality to your Activities. Services provide the ability to do such things as run scripts, schedule future events, send email, and receive email.

Services can be obtained from the Service Registry, which is found in the Interactive Spaces Environment, in two ways.

Services are obtained by giving a string name for the service. Examples of service names are `mail.sender` and `speech.synthesis`.

If you want to see if a service exists and your activity can run without it, you can use the `getService()` method of the Service Registry. This method will either return an instance of the requested service, or `null` if the service is not available.

```
Service service =
    getSpaceEnvironment().getServiceRegistry().getService("speech.synthesis");
```

If you need the service to exist and you want your activity to fail if the service is not available, you can use the `getRequiredService()` method of the Service Registry. if the service doesn't exist, the method will throw an `InteractiveSpacesException`.

```
Service service =
    getSpaceEnvironment().getServiceRegistry().getRequiredService("speech.synthesis");
```

16.1 The Script Service

The Script Service gives you the ability to script portions of your Activities. Interactive Spaces supports Python, Javascript, and Groovy out of the box. If you are running on a Mac, you will also have access to AppleScript.

The easiest way to use the Script Service is to give it the script as a string.

First get a Script Service from the Service Registry.

```
ScriptService scriptService =
    getSpaceEnvironment().getServiceRegistry().getService(ScriptService.SERVICE_NAME);
```

A very simple script would, in the time honored tradition, say *Hello, World*.

```
scriptService.executeSimpleScript("python", "print 'Hello, World'");
```

Sometimes you need to give values to your script from your Activity. You do that by providing a map with bindings in it. These bindings provide a set of variables in your script which it can use when it runs.

```
Map<String, Object> bindings = new HashMap<String, Object>();
bindings.put("message", "Hello, World");

scriptService.executeScript("python", "print message", bindings);
```

For more details about what you can do with the Script Service, see the ScriptService Javadoc.

16.2 Mail Sender Service

Interactive Spaces has the ability to send email through the mail Sender Service. With this service you can compose an email and then send it.

An example showing how to use the service is given below.

```
MailSenderService mailSenderService =
    getSpaceEnvironment().getServiceRegistry().getService(MailSenderService.SERVICE_NAME);

ComposableMailMessage message = new SimpleMailMessage();
message.setSubject("Greetings");
message.setFromAddress("the-space@interactivespaces.com");
message.addToAddress("you@you.com");
message.setBody("Hello World");

mailSenderService.sendMailMessage(message);
```

For more details about what you can do with the Mail Sender Service, see the MailSenderService Javadoc.

16.2.1 Configuring the Mail Sender Service

The Mail Sender Service needs to be configured properly if it going to be able to send mail. Configurations for the mail service should be placed in the `config/interactivespaces` directory. It is usually placed in a file called `mail.conf`.

The mail sender service needs to know an SMTP server which it can use to transport the mail to its destination. The SMTP server host is set with the `interactivespaces.service.mail.sender.smtp.host` configuration property. The port of the SMTP server is set with the `interactivespaces.service.mail.sender.smtp.port` configuration property.

An example would be

```
interactivespaces.service.mail.sender.smtp.host=172.22.58.11
interactivespaces.service.mail.sender.smtp.port=25
```

16.3 Mail Receiver Service

Interactive Spaces can also receive email through the Email Receiver Service. This service sets up a very simple SMTP server which can receive emails when properly configured. Event listeners are registered with the service which have methods which are called when an email is received.

n example showing how to use the service is given below.

```
MailReceiverService mailReceiverService =
    getSpaceEnvironment().getServiceRegistry().getService(MailReceiverService.SERVICE_NAME);

mailReceiverService.addListener(new MailReceiverListener() {
  public void onMailMessageReceive(MailMessage message) {
    getLog().info("Received mail from " + message.getFromAddress();
  }
});
```

The example listener merely prints the from address from the received email and nothing else.

A listener can me removed with the `removeListener()` method on the service.

For more details about what you can do with the Mail Receiver Service, see the MailReceiverService Javadoc.

16.3.1 Configuring the Mail Receiver Service

The Mail Receiver Service normally listens for SMTP traffic on port `9999`. It can be reconfigured. Configurations for mail services should be placed in the‘‘config/interactivespaces‘‘ directory, usually placed in a file called `mail.conf`.

The port of the SMTP receiver is set with the `interactivespaces.service.mail.receiver.smtp.port` configuration property.

An example would be

```
interactivespaces.service.mail.receiver.smtp.port=10000
```

16.4 Speech Synthesis Service

The Speech Synthesis Service allows your activities to speak. The service takes a string of text which is then spoken by the computer which contains the Space Controller running the service.

To use the Speech Synthesis Service, you must get an instance of a `SpeechSynthesisPlayer`. These players use various system resources which must be released when you are through with the player. One way of handling this would be to allocate a player in `onActivitySetup()` and add it as a Managed Resource for the Activity. Then Interactive Spaces will automatically clean up any resources used by the player when your activity stops running.

As an example, here is the first part of your Activity, showing the player instance variable and the code to obtain a player. Notice the player is added as a Managed Resource.

```
private SpeechSynthesisPlayer speechPlayer;

@Override
public void onActivitySetup() {
  SpeechSynthesisService speechSynthesisService =
      getSpaceEnvironment().getServiceRegistry().getRequiredService(
          SpeechSynthesisService.SERVICE_NAME);

  speechPlayer = speechSynthesisService.newPlayer();

  addManagedResource(speechPlayer);
}
```

Now making your activity speak is easy, you just use the `speak` method on the player.

```
speechPlayer.speak("Hello, world.", true);
```

The second argument for the `speak()` method determines if the method will block while the text is being spoken, or if it will return immediately with the text spoken asynchronously. if the value is `true` the method will block, if it is `false` the method will return immediately.

For more details about what you can do with the Speech Synthesis Service, see the SpeechSynthesisService Javadoc.

16.5 Chat Service

The Chat Service provides support for both reading from and writing to chat services. The current implementation only supports XMPP-based chat.

For more details about what you can do with the Chat Service, see the ChatService Javadoc.

16.6 Twitter Service

The Twitter Service provides support for both sending Twitter Status updates and being notified of any tweets containing a specified hashtag.

For more details about what you can do with the Chat Service, see the TwitterService Javadoc.

16.7 The Scheduler Service

The Scheduler Service gives you the ability to schedule some sort of task at some point in the future. These future tasks can be scheduled in a variety of ways, from one off events at some point in the future to tasks which repeat on schedules like every Monday, Wednesday, and Friday at 3AM.

For more details about what you can do with the Scheduler Service, see the SchedulerService Javadoc.

CHAPTER

SEVENTEEN

SUPPORT CLASSES

Interactive Spaces comes with a lot of builtin support for various common things you would want to do.

17.1 Persisted Maps

The *SimpleMapPersister* interface provides you with the ability to very simply persist a map of key/value pairs. The values can be lists or maps themselves. You can find detailed documentation in the SimpleMapPersister Javadoc. One particular implementation, *JsonSimpleMapPersister*, stores the maps as JSON files. You can find detailed documentation in the JsonSimpleMapPersister Javadoc.

Suppose that you want to create a map to persist in a subdirectory of the controller-wide data directory in an Activity. You can get that directory from the filesystem property of your Spaces Environment.

```
File basedir = getSpaceEnvironment().getFilesystem().getDataDirectory("maps");
SimpleMapPersister persister = new JsonSimpleMapPersister(basedir);

persister.putMap("foo", map);
```

Here *map* is assumed to be a map of data. Because the *JsonSimpleMapPersister* is being used, a file in *data/maps/foo.json* relative to where you installed your controller will be created with the contents of *map* serialized into the file as JSON.

```
Map<String, Object> map = persister.getMap("foo");
```

would then read the map from disk.

The SimpleMapPersister classes are thread-safe. They permit multiple simultaneous readers and only 1 writer. Any readers will block the writer and the writer will block any readers. The locks are fair, so continuing to add lots of readers will not block a writer forever.

CHAPTER

EIGHTEEN

THE INTERACTIVE SPACES COMM SUPPORT

Interactive Spaces supplies support for using serial and other communication interfaces to communicate with a variety of hardware. This lets you develop interactivity in your space using hardware like Arduinos.

18.1 Setting Up Serial Comm Support

The Serial Comm support does not come set up out of the box. It requires libraries specific to the platform running the Space Controller.

First locate a version of the RXTX Java Serial library for your platform. There are versions for Linux, MacOS, and Windows. Follow any directions found to download and install the library anywhere you want on the computer running the Space Controller.

The folder *extras* in your Space Controller installation contains two files

- interactivespaces-service-comm-serial-<version>.jar
- comm-serial-rxtx.ext

where <*version*> is the version of the Comm library found in the folder.

The JAR file should be copied into the *bootstrap* folder of the Space Controller. The *comm-serial-rxtx.ext* file should be copied into the Space Controller's *lib/system/java* folder. Edit the file and look for a line with the prefix *path:*. Change the rest of the line to point at the RXTX jar you have installed on the computer.

If the controller runs on a Linux device, the user which runs the controller must be in the *dialout* Linux group to use serial devices.

Now restart the Space Controller and the Serial Comm support will be available.

18.2 Using Serial Comm Support

18.2.1 An Example

An example of using the Serial Comm support can be found in the *interactivespaces.example.activity.arduino.analog.java* example in the Workbench's *examples* folder.

18.3 Setting Up Bluetooth Comm Support

The Bluetooth Comm support does not come set up out of the box. It requires libraries specific to the platform running the Space Controller.

First locate a version of the Bluecove Java library for your platform. There are versions for Linux, MacOS, and Windows. Follow any directions found to download and install the library anywhere you want on the computer running the Space Controller.

The folder *extras* in your Space Controller installation contains two files

- interactivespaces-service-comm-bluetooth-<version>.jar
- comm-bluetooth-bluecove.ext

where <*version*> is the version of the Bluetooth Comm library found in the folder.

The JAR file should be copied into the *bootstrap* folder of the Space Controller. The *comm-serial-bluecove.ext* file should be copied into the Space Controller's *lib/system/java* folder. Edit the file and look for a line with the prefix *path:*. Change the rest of the line to point at the Bluecove jar you have installed on the computer.

If the controller are running on Linux, you will need to add a second *path:* line to *comm-serial-bluecove.ext*. This line should point to the Bluecove GPL jar. Also, the user which runs the controller must be in the *lp* Linux group to use bluetooth devices.

Now restart the Space Controller and the Bluetooth Comm support will be available.

CHAPTER

NINETEEN

ADVANCED SPACE CONTROLLER USAGE

19.1 The *run* Folder

The folder where the Space Controller is installed contains a subfolder named *run*. This folder contains information about the running system and allows control of the Space Controller through files.

19.1.1 The PID File

The *run* folder contains a file called *interactivespaces.pid*. This file gives the Process ID (or PID) of the operating system process the controller is running under. The file contains only the Process ID number with nothing else, including no end of line characters.

This file should not exist when the Space Controller starts. If the Space Controller tries to start and this file exists, the Space Controller will complain about the existence of the file and shut itself down. This prevents the Space Controller from having multiple instances running at the same time from the same installation. If two Space Controllers are needed on the same host computer, two separate installs must be done on that Host Computer and each Space Controller must have a different Host ID and UUID.

The PID file is deleted when the Space Controller is cleanly shut down. Should the Space Controller crash, the PID file will be left and must be deleted before the Space Controller can be started again.

19.1.2 The *control* Folder

The *run* folder can contain a subfolder called *control*. This subfolder allows control of the Space Controller by files which are added to it.

For example, creating a file called *shutdown* in this folder will shut the controller down softly. All running Live Activities will be cleanly shutdown and then the entire Space Controller will exit.

The *control* folder can be created after the Space Controller has been started.

CHAPTER

TWENTY

ANDROID

You can run Interactive Spaces Controllers on Android devices, including phones, tablets, and Google TV devices. This permits you to control applications on the Android device, access the cloud services that Android makes possible, and gives you access to a wide variety of sensors which come with most devices.

20.1 Overview

The Android-based Interactive Spaces Controller provides much of the same experience as a desktop-based controller. You author activities, compile them with the Interactive Spaces workbench, load them into your local Interactive Spaces Master, deploy them to the Android controller, and control their lifecycle from the master. For many operations, the Android Controller will work the same as desktop controllers.

20.1.1 Technical Mumbo Jumbo

The Android Controller runs as an Android service. This means it it is still running behind the scenes, even if other Android applications currently have the screen. A foreground service is used, similar to a music player, which means that the Controller runs at a high priority and should not be killed by Android as it manages the often limited memory in the device.

20.1.2 Limitations

There are limitations to the Android Controller due to the nature of Android. It is possible to dynamically update desktop Controllers, allowing you to upgrade or extend Interactive Spaces itself while the Controller is running. This is not possible with Android, to make everything work portions of the Controller had to be permanently compiled into the Android application and updates will require updating the Android application itself.

Also, some of the services available on desktop Controllers are not available on Android, most notably the script service. This means that Android Interactive Spaces Activities can not be implemented as scripts, but must be implemented as Java applications.

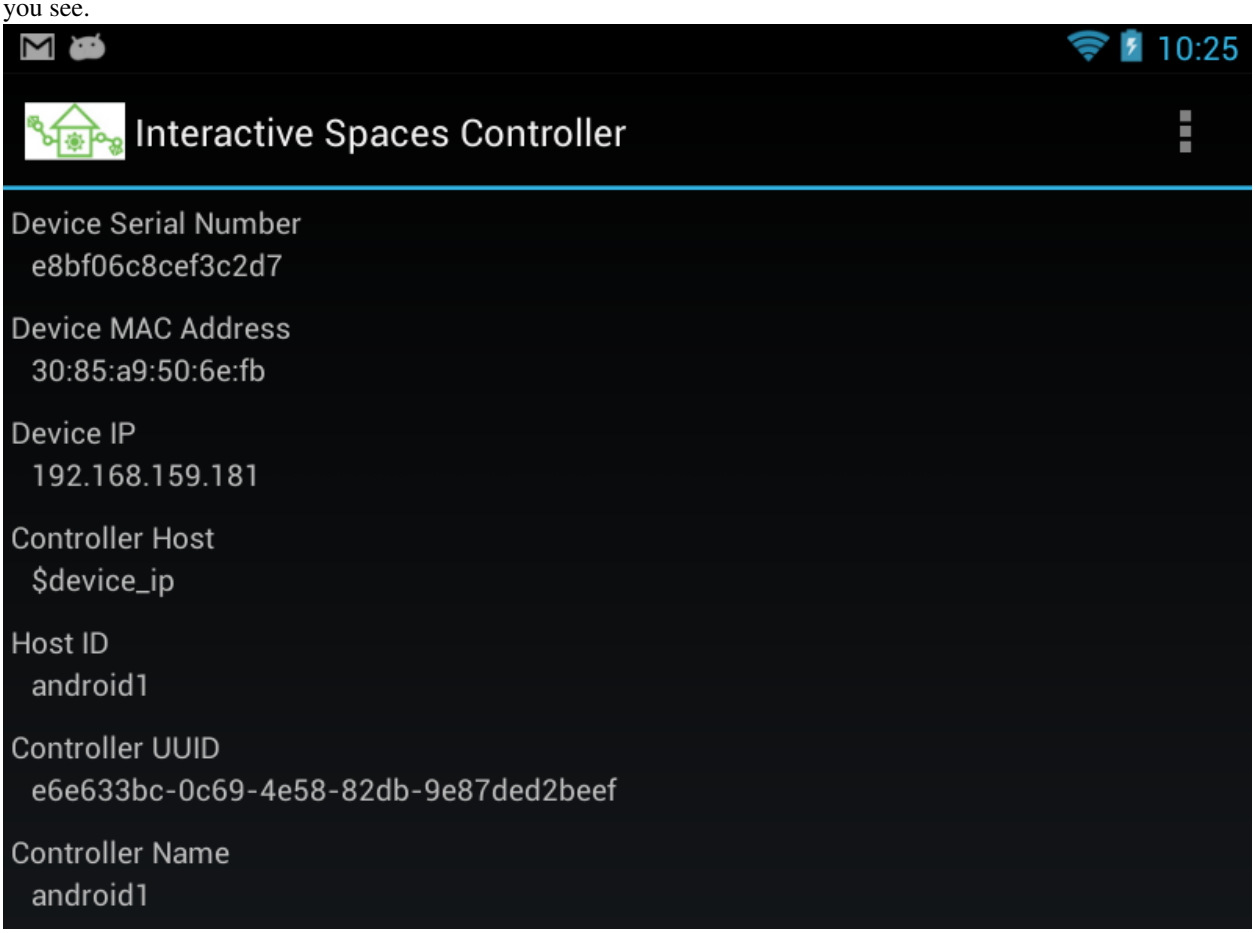
20.2 Setup

If you want to use precompiled Android Interactive Spaces activities, all you need to do is install the Android Interactive Spaces Controller, configure it, and then start it up.

20.2.1 Installing the Controller

The Interactive Spaces release has a file named *interactivespaces-controller-android.apk*. This is the Android application which must be loaded onto your Android device. You will need to sideload this application into your device.

Once you have the Controller installed, start the application running. What you will see is the information page for the controller, part of which can be seen below. Any time you run the Controller application, this will be the screen you see.

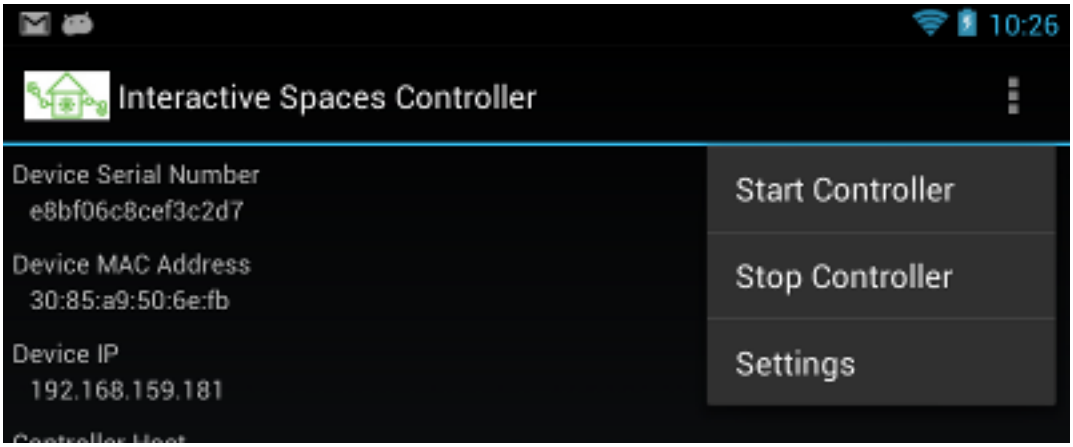


Some of the fields are purely informational, like the *Device Serial Number* and the *Device MAC Address*. Others are settings that you must configure before your first use of the controller. Attempting to start the controller before it is properly configured will cause a message to appear on this screen.

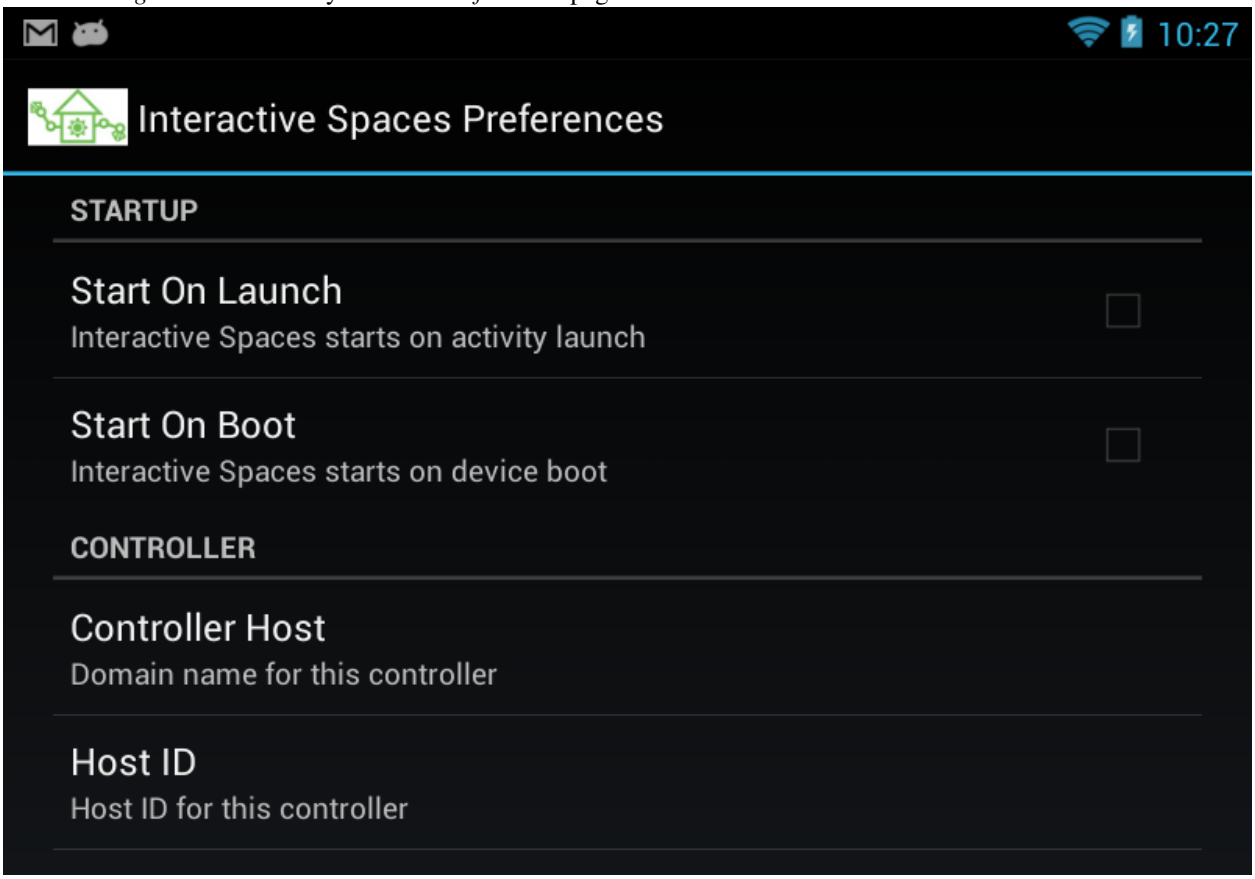
20.2.2 Configuring the Controller

You will only need to configure a given Controller once.

To get to the preferences screen, open the Controller menu.



Select *Settings*. This will take you to the *Preferences* page.



The only Preferences you must set are the

- Host ID
- Name
- Master Host

Remember, the *Name* is the name of the controller which will appear in the Master's lists of controllers and can be anything you want.

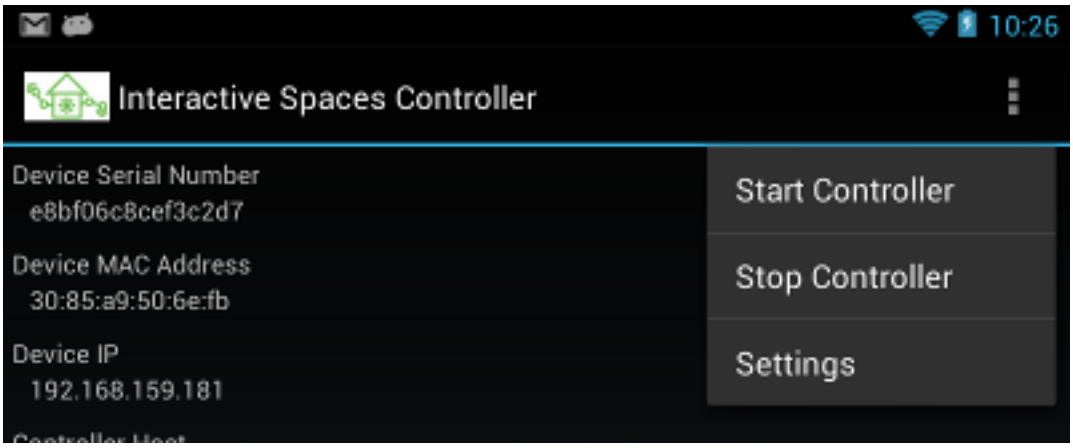
The *Controller Host* field is the domain name that the Controller will use when different components of your Interactive Spaces installation communicate with your controller and the Live Activities that run on it. If your device gets its

domain name from a DHCP server, that name should be used. If your device has a fixed domain name on your network, that name should be used. This field can also use the magic value *$device_ip*, which means that the domain name for the device will be its IP address. This will work well for initial development, but having a real domain name for your devices in your Interactive Spaces environment is preferred.

Make sure you have no fields on the main screen which say *?required?*.

20.2.3 Starting Up the Controller

The Android Controller is started up by using the *Start Controller* menu item on the Controller menu.



If you have set all required fields in the Preferences, you should see the Interactive Spaces logo on your notification bar. If any fields are not filled out you should see a box telling you what has not been properly set.

You can also have the Controller immediate start whenever the Controller application is started, when you see the information screen, or even when the Android device boots. You can set this with the *Startup* Preferences on the preferences page.

20.2.4 Setup for Writing Android Activities

There are several things that must be done before you can create Android Interactive Spaces Activities. The Workbench must be told where your Android SDK is so that you can compile.

To configure the Workbench, go to the folder where you installed the Workbench. Inside the Workbench's *config* folder is a file called *android.conf*.

Find the root folder for your Android SDK. This folder will have folders with names like *platform-tools* and *platforms*. The path to this folder will be the value for the *android.sdk.home* property in *android.conf*.

if you look inside the *platforms* folder of this root folder, you will see a bunch of folders which give names of Android versions. For example, on myMac I see *android-4.2*, and on my Linux box I see things like *android-11* and *android-17*. The *android.platform* property in the *android.conf* should be set the the name of one of these folders.

This will be the Android version for all Android Activities you write with the Workbench. Eventually you will be able to pick the Android SDK version supported from the project, but for now you have to pick a global value.

20.3 Writing Android-based Activities

You can create and build Android-based Activities using the Interactive Spaces Workbench. When creating the project, be sure to specify the language as *android*.

```
java -jar interactivespaces-launcher-0.0.0.jar create language android
```

if using the command line interface, or by chosing one of the Android templates if using the Workbench GUI.

20.3.1 The AndroidOs Service

For the most part, you can use many of the Interactive Spaces Activity support classes for implementing your applications. Browser-based web applications will automatically start the browser for you, so you can use the standard web Activity base classes or Activity Components.

But sometimes you need more direct access to Android services. You can do this by getting access to the AndroidOs service. This will give you the service context that the Controller is running as, which allows you access to sensors, activity startup through Intents, etc.

During Activity setup, you can obtain the AndroidOS service through the Service Registry.

```
AndroidOsService androidService =
    getSpaceEnvironment().getServiceRegistry().getService(AndroidOsService.SERVICE_NAME);
```

You can then get the context and Android services from the service.

20.4 Examples

20.4.1 Launching An Android Activity

Launching an Android activity is quite simple. You obtain the Android context for the Interactive Spaces Controller and use it to launch the Android activity (it is unfortunate that Interactive Spaces has activities and Android has activities as well, it makes for confusing sentences).

The following example is from the workbench. Note the use of *Intent.FLAG_ACTIVITY_NEW_TASK*, which is required to start up an Android activity outside of the Interactive Spaces Controller.

```
public class SimpleAndroidWebActivity extends BaseActivity {

    @Override
    public void onActivityStartup() {
        AndroidOsService androidService = getSpaceEnvironment()
             .getServiceRegistry().getService(AndroidOsService.SERVICE_NAME);

        try {
            Intent browserIntent = new Intent(Intent.ACTION_VIEW );
            browserIntent.setData( Uri.parse("https://code.google.com/p/interactive-spaces/") );
            browserIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            androidService.getAndroidContext().getApplicationContext().startActivity(browserIntent);
        } catch (Exception e) {
            getLog().error("Unable to start browser", e);
        }
    }
}
```

20.4.2 Reading the Android Accelerometer

As an example for sensor use, here is the Accelerometer Activity from the Workbench examples. The AndroidOS service is used to get the Android Sensor Manager. The accelerometer is obtained from the Sensor Manager.

Notice the accelerometer is released in the cleanup event.

```
public class AccelerometerAndroidActivity extends BaseRoutableRosActivity {

    private SensorManager sensorManager;
    private Sensor accelerometer;
    private SensorEventListener accelerometerEventListener;

    @Override
    public void onActivitySetup() {
        getLog().info(
                "Activity interactivespaces.example.activity.android.accelerometer setup");

        AndroidOsService androidService = getSpaceEnvironment()
                .getServiceRegistry().getService(AndroidOsService.SERVICE_NAME);
        sensorManager = (SensorManager) androidService
                .getSystemService(Context.SENSOR_SERVICE);
        accelerometer = sensorManager
                .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        accelerometerEventListener = new SensorEventListener() {

            @Override
            public void onAccuracyChanged(Sensor sensor, int accuracy) {
                // Nothing yet
            }

            @Override
            public void onSensorChanged(SensorEvent event) {
                onAccelerometerEvent(event);
            }
        };
        sensorManager.registerListener(accelerometerEventListener,
                accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    public void onActivityCleanup() {
        sensorManager.unregisterListener(accelerometerEventListener);
    }

    private void onAccelerometerEvent(SensorEvent event) {
        if (isActivated()) {
            // Do stuff...
        }
    }
}
```

CHAPTER

TWENTYONE

THE EXPRESSION LANGUAGE

The Interactive Spaces Expression Language is used in a variety of places, from the Activity configurations to the Master API. It is pretty easy to use and is based on the Apache OGNL expression language.

21.1 Accessing Arrays and Maps

You can access an array with *[]*. For example, suppose there is an array called *foo*. *foo[0]* will get the first element in the array.

The same syntax is used for maps. For example, suppose there is a map called *metadata*. *metadata['author']* will get the map value with key *author*. If there is no value associated with the given key, the expression will be equal to *null*.

21.2 Conditionals

e1 ? e2 : e3 is the conditional operator. *e1* is first evaluated. If its value is *true*, the value of the expression will be *e2*. Otherwise, it will be *e3*.

You can check the equality of two expressions using the == or *eq* operators.

CHAPTER

TWENTYTWO

ADVANCED MASTER USAGE

22.1 The *run* Folder

The folder where the Master is installed contains a subfolder named *run*. This folder contains information about the running system and allows control of the Master through files.

22.1.1 The PID File

The *run* folder contains a file called *interactivespaces.pid*. This file gives the Process ID (or PID) of the operating system process the Master is running under. The file contains only the Process ID number with nothing else, including no end of line characters.

This file should not exist when the Master starts. If the Master tries to start and this file exists, the Master will complain about the existence of the file and shut itself down. This prevents the Master from having multiple instances running at the same time from the same installation.

The PID file is deleted when the Master is cleanly shut down. Should the Master crash, the PID file will be left and must be deleted before the Master can be started again.

22.1.2 The *control* Folder

The *run* folder can contain a subfolder called *control*. This subfolder allows control of the Master by files which are added to it.

Creating a file called *shutdown* in this folder will shut the master down softly.

Creating a file called *space-controllers-shutdown-all* will shut down all Space Controllers the Master knows about.

Creating a file called *space-controllers-shutdown-all-activities* will shut down all Live Activities running on all Space Controllers the Master knows about.

Creating a file called *live-activity-group-startup-id* start up a Live Activity Group whose ID is *id*. For example, *live-activity-group-startup-652* will start up Live Activity Group *652*.

Creating a file called *live-activity-group-activate-id* start up a Live Activity Group whose ID is *id*. For example, *live-activity-group-activate-652* will activate Live Activity Group *652*.

Creating a file called *space-startup-id* start up a Space whose ID is *id*. For example, *space-startup-652* will start up Space *652*.

Creating a file called *space-activate-id* start up a Space whose ID is *id*. For example, *space-activate-652* will activate Space *652*.

Creating a file called *script-run-id* run a script whose ID is *id*. For example, *script-run-652* will run the script with ID *652*.

The *control* folder can be created after the Master has been started.

This kind of control of the Master is useful for automated starts and stops of the Master. A job scheduling system on the Master's host computer, for example CRON on a Linux box, can start the Master up before it is needed. A second job could then write *shutdown* into the control when the master is no longer needed.

Space Controllers are not automatically shut down if the Master is shut down. Stopping the Master while Space Controllers are running is currently not supported, make sure you shut down all Space Controllers before shutting the Master down. The SpaceOperations script helps with this task.

22.2 Automatic Activity Import

Activities can be autoimported in to the master.

To use this, you must create a folder called *master/activity/import* in the same folder where your Master is installed.

Now any Activity zip files which are placed in this folder will be auto-imported into the master. If there is already an Activity with the same identifying name and version, it will be replaced with this new Activity.

The Live Activities based on this Activity will not be immediately updated. You must do that manually using the Master webapp or Master API.

Activities can be autoimported in to the master and then deployed.

To use this, you must create a folder called *master/activity/deploy* in the same folder where your Master is installed.

Now any Activity zip files which are placed in this folder will be auto-imported into the master and will then be immediately deployed to all controllers which contain a Live Activity based on the imported activity. If there is already an Activity with the same identifying name and version, it will be replaced with this new Activity.

22.3 Scripting the Master

It is possible to write small scripts which can instruct the master to do a variety of tasks. The Interactive Spaces scripting service currently supports Javascript, Python, and Groovy.

22.3.1 An Example Script

The following is an example of a Python script which will get all Live Activities that the master knows about and deploys them.

```
for liveActivity in activityRepository.allLiveActivities:
  print "%s: %s" % (liveActivity.id, liveActivity.name)
  uiControllerManager.deployLiveActivity(liveActivity.id)
```

22.3.2 Named Scripts

The Master also supports Named Scripts, which are scripts stored in the master database. These scripts are run from the Master Webapp or the Master API. They can also be scheduled to run via the scheduler.

22.3.3 Startup Master Extensions

One way to script the Master is through the use of Startup Master Extensions. These extensions are run after the master starts up.

During startup, the Master will look in the folder *extensions/startup* in the same folder where your master is installed. These files will then be run in sorted order alphabetically by name.

For instance, if the extensions folder contains *011-foo.groovy* and *001-bar.py*. They will be run in the order

1. 001-bar.py
2. 011-foo.groovy

Any extensions added after the master is started will be run immediately. They will then be run in the name sorted order next time the Master is started.

So say you add *05-banana.groovy* to the extensions folder. It will be run immediately. But next time the master starts, the order will be

1. 001-bar.py
2. 005-banana.groovy
3. 011-foo.groovy

22.3.4 API Master Extensions

API Master Extensions allow you to add special extensions to the Master WebSocket API.

The Master looks for API Extensions in the folder *extensions/api* in the same folder where your master is installed. Extensions can be added to this folder before the Master is started and while it is running.

For the first example, suppose you have the file *extensions/api/settings-get.groovy*, which is a Groovy based script. You could call it with the following web socket call.

```
{command: '/extension/settings-get', args: {map: 'b'}}
```

The script in *extensions/api/settings-get.groovy* could be something like

```
def map = spaceEnvironment.getValue('master.settings.map')
if (map) {
  [result: "success", data: map.getMap(args.map)]
} else {
  [result: "failure", reason: "no map"]
}
```

This script is written to get a *SimpleMapPersister* named *master.settings.map* in the Space Environment. If the map is there, the Script returns the map with the name *args.map*, which, in the example call given above, would have a value of *b*. *args* is a map of arguments for the call. The *b* map would then be sent over the web socket channel. If the persister doesn't exist, a map giving a failure result would be returned.

Then suppose there was a script called *extensions/api/settings-put.groovy* which is called with the following command

```
{ command: '/extension/settings-put', args: {map: 'b', data: {e: 'f', g: 'h'}}}
```

with the script contents being

```
def map = spaceEnvironment.getValue('master.settings.map')
if (map) {
  map.putMap(args.map, args.data)
```

```
  [result: "success"]
} else {
  [result: "failure", reason: "no map"]
}
```

Here we get the same persisted map from the previous example, map *b*, and put the data *{e: 'f', g: 'h'}* into the map.

22.3.5 System Objects Available

Functionality for controlling the master is found in a collection of repositories which store the various entities the master understands, and managers which can perform operations like deploying a Live Activity or starting up a Live Activity Group.

The Scripting Service

The Scripting Service allows you to run scripts in the master in a variety of languages.

The service will be called *scriptService* in your script.

You can find detailed documentation in the ScriptService Javadoc.

The Scheduler Service

The Scheduler Service allows you to schedule tasks in the master.

The service will be called *schedulerService* in your script.

You can find detailed documentation in the SchedulerService Javadoc.

The Controller Repository

The Controller Repository contains all known space controllers.

The service will be called *controllerRepository* in your script.

You can find detailed documentation in the ControllerRepository Javadoc.

The Activity Repository

The Activity Repository contains all known activities, live activities, and live activity groups.

The service will be called *activityRepository* in your script.

You can find detailed documentation in the ActivityRepository Javadoc.

The Space Repository

The Space Repository contains all known Spaces.

The service will be called *spaceRepository* in your script.

You can find detailed documentation in the SpaceRepository Javadoc.

The Active Controller Manager

The Active Controller manager is used to control the Alive Activities on a remote Space Controller.

The service will be called *activeControllerManager* in your script.

You can find detailed documentation in the ActiveControllerManager Javadoc.

The UI Activity Manager

The UI Activity Manager is used to perform various operations on Activities. It is a UI Manager as it only requires a few arguments, like an Activity ID, rather than an actual domain object.

The service will be called *uiActivityManager* in your script.

You can find detailed documentation in the UiActivityManager Javadoc.

The UI Controller Manager

The UI Controller Manager is used to perform various operations on Space Controllers, including the Live Activities they contain. It is a UI Manager as it only requires a few arguments, like a Space Controller ID or a Live Activity ID, rather than an actual domain object.

The service will be called *uiControllerManager* in your script.

You can find detailed documentation in the UiControllerManager Javadoc.

The UI Master Support Manager

The UI Master Support Manager is used for advanced support of the manager. This includes such operations as getting and importing a Master Domain model which describes every aspect of the space.

The service will be called *uiMasterSupportManager* in your script.

You can find detailed documentation in the UiMasterSupportManager Javadoc.

The Interactive Spaces Environment

The Interactive Spaces Environment is a hook into the guts of Interactive Spaces for the master. It gives access to logs, the container filesystem, and many other aspects of the container.

The service will be called *spaceEnvironment* in your script.

You can find detailed documentation in the InteractiveSpacesEnvironment Javadoc.

The Automation Manager

The Automation Manager is used for automating tasks within the Master. It gives another way of accessing the scripting service and easily running a script in a variety of languages.

The service will be called *automationManager* in your script.

You can find detailed documentation in the AutomationManager Javadoc.

22.4 Moving Ports for the Master

Sometimes you might not be able to use the default ports that the Interactive Spaces Master uses.

The Master contains a ROS master used by the core communication facilities provided by Interactive Spaces. The file *config/container.conf* contains a line like

```
org.ros.master.uri=http://masterhost:11311/
```

where *masterhost* is the host name for the machine the Master is running on. The port, here *11311*, can be changed on this line to any other port. For example, if the ROS master should run on port *11312*, this line should become

```
org.ros.master.uri=http://masterhost:11312/
```

The Master Web Application's port, *8080* by default, can be changed with the configuration property *org.osgi.service.http.port*. This property is set in *config/container.conf*.

The Master uses an HTTP server for deploying Live Activities to their controllers. The controller receives a URL for this server when the Master tells it a Live Activity is being deployed to the controller. The port for this HTTP server can be changed with the configuration property *interactivespaces.repository.activities.server.port*. The default value of *10000* is used if this property doesn't exist. This configuration property should be set in *config/interactivespaces/master.conf*.

The Master uses a websocket server for live activities that want to provide a richer admin interface than the Master Web Application. The port for this websocket server can be changed with the configuration property *interactivespaces.master.api.websocket.port*. The default value of *8090* is used if this property doesn't exist. This configuration property should be set in *config/interactivespaces/master.conf*.

22.5 Notification for Issues

The Space Controllers are constantly sending a heartbeat back to the Master so that the master knows the Space Controllers are still connected and alive. If a Space Controller dies or loses network connectivity, it is possible to receive an alert.

22.5.1 Email Alerts

The only alert mechanism available out of the box is an email-based one. The alert mechanism will send an email containing information about the alert to a group of email addresses.

The email alert mechanism is configured through the file *config/mail.conf*. A sample file is given below.

```
interactivespaces.mail.smtp.host=192.168.172.12
interactivespaces.mail.smtp.port=25

interactivespaces.service.alert.notifier.mail.to = person1@foo.com person2@foo.com
interactivespaces.service.alert.notifier.mail.from = interactivespaces@foo.com
interactivespaces.service.alert.notifier.mail.subject = Death, doom, and destruction in My Space
```

The property *interactivespaces.mail.smtp.host* specifies a host running an SMTP server which will relay the alert. The' property *interactivespaces.mail.smtp.port* can be used to specify the port this SMTP server is listening on.

The property *interactivespaces.service.alert.notifier.mail.to* specifies who should receive the alert email. The recipient email addresses on this list are separated by spaces or tabs, and there can be as many addresses as are needed.

The property *interactivespaces.service.alert.notifier.mail.from* specifies what the From address of the email will be.

The property *interactivespaces.service.alert.notifier.mail.subject* gives the Subject line the alert email will have.

A sample email, though the format is subject to change, for losing contact with a Space Controller is

```
No space controller heartbeat in 30881 milliseconds

ID: 56
UUID: 83aab854-ead1-482e-8ce5-0fcca7b508e8
Name: The Living Room Controller
HostId: livingroomcontroller
```

CHAPTER

TWENTYTHREE

THE MASTER API

The Master exposes a web-based API which allows you to write your own web applications to control the master. The interface is mostly REST-ful, and returns results in JSON.

23.1 Common Features of the API

The API has several features which are common to all methods

23.1.1 API URL

Every API URL fill be prefixed with the following

http://spacemaster:8080/interactivespaces/

where *spacemaster* is the domain name of the machine running the Interactive Spaces Master.

For example, the API call to obtain all Live Activities know by the system is

http://spacemaster:8080/interactivespaces/liveactivity/all.json

23.1.2 Results

The JSON results returned have features in common.

Success

The JSON result for a successful API command which has no return data will be

```
{
  "result": "success"
}
```

If there is data to be returned, the results will be

```
{
  "result": "success",
  "data": result data
}
```

where *result data* will be a JSON object giving the result.

Failure

The JSON result for a failed API command will be

```
{
  "result": "failure",
  "message": content
}
```

where *content* will be a description of what failed.

23.2 Space Controllers

The Space Controller API allows you to get a collection of all Space Controllers known by the Master. Once you have the IDs, you can then connect and disconnect from them.

23.2.1 Getting All Space Controllers

The Master API call to get the list of all Space Controllers is

```
/spacecontroller/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response of the following form

```
{
  "id": id,
  "uuid": uuid,
  "name": name,
  "description": description,
}
```

The names and a description of their values is given below.

The information includes a small amount of information about the Activity portion of the Live Activity.

id	The ID of the Controller
uuid	The UUID of the Controller
name	The name of the Controller
description	The description of the Controller

23.2.2 Getting the Status of All Space Controllers

The full status from all space controllers is requested by the API command

```
/spacecontroller/all/status.json
```

The JSON returned will the simple JSON success result.

23.2.3 Connecting to a Space Controller

A Space Controller is connected to by the API command

```
/spacecontroller/id/connect.json
```

where *id* is the ID of the Controller. Be sure you use the ID of the Controller, not the UUID.

The JSON returned will the simple JSON success result.

23.2.4 Disconnecting from a Space Controller

A Space Controller is disconnected from the master by the API command

`/spacecontroller/id/disconnect.json`

where *id* is the ID of the Controller. Be sure you use the ID of the Controller, not the UUID.

The JSON returned will the simple JSON success result.

23.2.5 Shutting Down All Activities on a Space Controller

All Live Activities on Space Controller can be shut down by the API command

`/spacecontroller/id/activities/shutdown.json`

where *id* is the ID of the Controller. Be sure you use the ID of the Controller, not the UUID.

The JSON returned will the simple JSON success result.

23.2.6 Shutting Down a Space Controller

A Space Controller can be remotely shut down by the API command

`/spacecontroller/id/shutdown.json`

where *id* is the ID of the Controller. Be sure you use the ID of the Controller, not the UUID.

The JSON returned will the simple JSON success result.

23.2.7 Deploying all Known Live Activities

All known Live Activities on the controller are deployed by the API command

`/spacecontroller/id/deploy.json`

where *id* is the ID of the Controller. Be sure you use the ID of the Controller, not the UUID.

The JSON returned will the simple JSON success result.

23.3 Activities

The Activities API allows you to get a collection of all Activities known by the Master. Once you have the IDs, you can then deploy all known Live Activity instances using that Activity.

23.3.1 Getting All Activities

Suppose the Master is running on your local host. The URL to get the list of all Activities is

```
/activity/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response of the following form

```
{
  "id": id,
  "identifyingName": identifyingName,
  "version": version,
  "name": name,
  "description": description,
  "lastUploadDate", lastUploadDate,
  "metadata": metadata
}
```

The names and a description of their values is given below.

id	The ID of the Activity
identifyingName	The identifying name of the Activity
version	The version of the Activity
name	The name of the Activity
description	The description of the Activity
metadata	The metadata of the Activity
lastUploadDate	The number of milliseconds since January 1, 1970 that the Activity was last uploaded

You can add a query parameter called *filter* onto the URL which will restrict the activities returned. For details on how to write the filters, For details on the expression language that you write filters in, see *The Expression Language*. The context of the filter will be the activity itself, so you can refer to any of the above properties directly.

For example

```
metadata['author'] eq 'Keith Hughes'
```

will return all activities if its metadata contains an *author* field with *Keith Hughes* as its value.

23.3.2 Deploying a Activity

All out of date Live Activity instances of a Activity are deployed by the API command

```
/activity/id/deploy.json
```

where *id* is the ID of the Activity.

The JSON returned will the simple JSON success result.

23.4 Live Activities

The Live Activities API allows you to get a collection of all Live Activities known by the Master. Once you have the IDs, you can then deploy, configure, start, stop, activate, deactivate, and get the status on all live activities.

23.4.1 Getting All Live Activities

Suppose the Master is running on your local host. The URL to get the list of all Live Activities is

```
/liveactivity/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response where each entry in the list will be of the form

```
{
  "id": id,
  "uuid": uuid,
  "name": name,
  "description": description,
  "status" : status,
  "statusMessage" : statusMessage,
  "metadata" : metadata
  "activity": {
    "identifyingName": identifyingName,
    "version": version,
    "metadata": activityMetadata
  },
  "controller": {
    "id": controllerId,
    "name": controllerName
  }
}
```

The names and a description of their values is given below.

The information includes a small amount of information about the Activity portion of the Live Activity.

id	The ID of the Live Activity
uuid	The UUID of the Live Activity
name	The name of the Live Activity
description	The description of the Live Activity
metadata	The metadata of the Live Activity
identifyingName	The identifying name of the Activity
version	The version of the Activity
activityMetadata	The metadata of the Activity
controllerId	The ID of the controller
controllerName	The name of the controller

See *Getting the Status of a Live Activity* for details on the status fields.

You can add a query parameter called *filter* onto the URL which will restrict the activities returned. For details on how to write the filters, For details on the expression language that you write filters in, see *The Expression Language*. The context of the filter will be the activity itself, so you can refer to any of the above properties directly.

For example

```
metadata['author'] eq 'Keith Hughes'
```

will return all live activities whose metadata contains an *author* field with *Keith Hughes* as its value.

23.4.2 Viewing a Live Activity

You can get the basic information for a Live Activity by the API command

```
/liveactivity/id/view.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

This returns JSON in the *data* portion of a successful JSON response of the form

```
{
  "id": id,
  "uuid": uuid,
  "name": name,
  "description": description,
  "status" : status,
  "statusMessage" : statusMessage,
  "metadata" : metadata
  "activity": {
    "identifyingName": identifyingName,
    "version": version,
    "metadata": activityMetadata
  },
  "controller": {
    "id": controllerId,
    "name": controllerName
  }
}
```

The names and a description of their values is given below.

The information includes a small amount of information about the Activity portion of the Live Activity.

id	The ID of the Live Activity
uuid	The UUID of the Live Activity
name	The name of the Live Activity
description	The description of the Live Activity
metadata	The metadata of the Live Activity
identifyingName	The identifying name of the Activity
version	The version of the Activity
activityMetadata	The metadata of the Activity
controllerId	The ID of the controller
controllerName	The name of the controller

See *Getting the Status of a Live Activity* for details on the status fields.

23.4.3 Configuring a Live Activity

The configuration for a Live Activity is sent to the remote installation by the API command

```
/liveactivity/id/configure.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.4 Getting the Configuration of a Live Activity

The configuration for a Live Activity is obtained by the API command

```
/liveactivity/id/configuration.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

This returns a JSON map in the *data* portion of a successful JSON response. The map will be keyed by the name of a configuration parameter. The map value will be the value for the configuration parameter.

```
{
  "param1": "value of param 1",
  "param2": "value of param 2"
}
```

23.4.5 Setting the Configuration of a Live Activity

The configuration for a Live Activity be set by the API command

```
/liveactivity/id/configuration.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

This must be a POST call with type *application/json*. The body of post should be a JSON map where the keys are the names of configuration parameters and the values will be the value of the associated parameter.

```
{
  "param1": "value of param 1",
  "param2": "value of param 2"
}
```

The JSON returned will the simple JSON success result.

23.4.6 Deploying a Live Activity

A Live Activity is deployed by the API command

```
/liveactivity/id/deploy.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.7 Starting Up a Live Activity

A Live Activity is started up by the API command

```
/liveactivity/id/startup.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.8 Activating a Live Activity

A Live Activity is activated by cthe API command

```
/liveactivity/id/activate.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.9 Deactivating a Live Activity

A Live Activity is deactivated by calling the API command

```
/liveactivity/id/deactivate.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.10 Shutting Down a Live Activity

A Live Activity is shut down calling the API command

```
/liveactivity/id/shutdown.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON returned will the simple JSON success result.

23.4.11 Getting the Status of a Live Activity

The status of a Live Activity is obtained by calling the API command

```
/liveactivity/id/status.json
```

where *id* is the ID of the Live Activity. Be sure you use the ID of the Live Activity, not the UUID.

The JSON success result with a *data* field which contains the following result.

```
{
  "status" : status,
  "statusMessage" : statusMessage
}
```

status will be one of the following.

space.activity.state.unknown

The status is unknown

space.activity.state.deployment.attempt

A deployment is being attempted

space.activity.state.deployment.failure

A deployment attempt has failed

space.activity.state.ready

The Live Activity is ready to to run

space.activity.state.start.attempt

A startup is being attempted

space.activity.state.start.failure

A startup attempt has failed

space.activity.state.running

The Live Activity is running

space.activity.state.activate.attempt

An activation is being attempted

space.activity.state.activate.failure

An activation attempt has failed

space.activity.state.active

The Live Activity is active

space.activity.state.deactivate.attempt

A deactivation is being attempted

space.activity.state.deactivate.failure

A deactivation attempt has failed

space.activity.state.shutdown.attempt

A shutdown is being attempted

space.activity.state.shutdown.failure

A shutdown attempt has failed

space.activity.state.crash

The Live Activity has crashed

statusMessage will be the status in a more human-readable format.

23.5 Live Activity Groups

The Live Activity Groups API allows you to get a collection of all Live Activity Groups known by the Master. Once you have the IDs, you can then deploy, configure, start, stop, activate, and deactivate all Groups.

23.5.1 Getting All Live Activity Groups

The API call to get the list of all Live Activity Groups is

```
/liveactivitygroup/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response of the following form

```
{
  "id": id,
  "name": name,
  "description": description,
  "metadata", metadata
}
```

The names and a description of their values is given below.

id	The ID of the Group
name	The name of the Group
description	The description of the Group
metadata	The metadata of the Group

You can add a query parameter called *filter* onto the URL which will restrict the Groups returned. For details on how to write the filters, For details on the expression language that you write filters in, see *The Expression Language*. The context of the filter will be the Group itself, so you can refer to any of the above properties directly.

For example

```
metadata['author'] eq 'Keith Hughes'
```

will return all Groups whose metadata contains an *author* field with *Keith Hughes* as its value.

23.5.2 Viewing a Live Activity Group

The URL to get information about a specific Live Activity Group is

```
/liveactivitygroup/id/view.json
```

where *id* is the ID of the Group.

This returns a JSON object in the *data* portion of a successful JSON response of the form

```
{
  "id": id,
  "name": name,
  "description": description,
  "liveActivities": liveActivities,
}
```

The names and a description of their values is given below.

id	The ID of the Group
name	The name of the Group
description	The description of the Group
metadata	The metadata of the Group

The *liveActivities* field will be a list of information for each Live Activity in the Group. See *Getting All Live Activities* to see the data that will be given for each Live Activity.

23.5.3 Requesting the Status of all Live Activities a Live Activity Group

A request to get a status update of all Live Activities in the Group can be initiated by the API command

```
/liveactivitygroup/id/liveactivitystatus.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.5.4 Deploying a Live Activity Group

A Live Activity Group is deployed by the API command

```
/liveactivitygroup/id/deploy.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.5.5 Starting Up a Live Activity Group

A Live Activity Group is started up by the API command

```
/liveactivitygroup/id/startup.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.5.6 Activating a Live Activity Group

A Live Activity Group is activated by cthe API command

```
/liveactivitygroup/id/activate.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.5.7 Deactivating a Live Activity Group

A Live Activity Group is deactivated by calling the API command

```
/liveactivitygroup/id/deactivate.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.5.8 Shutting Down a Live Activity Group

A Live Activity Group is shut down calling the API command

```
/liveactivitygroup/id/shutdown.json
```

where *id* is the ID of the Group.

The JSON returned will the simple JSON success result.

23.6 Spaces

The Spaces API allows you to get a collection of all Spaces known by the Master. Once you have the IDs, you can then deploy, configure, start, stop, activate, and deactivate all Spaces.

23.6.1 Getting All Spaces

The API call to get the list of all Spaces is

```
/space/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response of the following form

```
{
  "id": id,
  "name": name,
  "description": description,
  "metadata", metadata
}
```

The names and a description of their values is given below.

id	The ID of the Space
name	The name of the Space
description	The description of the Space
metadata	The metadata of the Space

You can add a query parameter called *filter* onto the URL which will restrict the Spaces returned. For details on how to write the filters, For details on the expression language that you write filters in, see *The Expression Language*. The context of the filter will be the activity itself, so you can refer to any of the above properties directly.

For example

```
metadata['author'] eq 'Keith Hughes'
```

will return all Spaces whose metadata contains an *author* field with *Keith Hughes* as its value.

23.6.2 Viewing a Space

The URL to get information about a specific Space is

```
/space/id/view.json
```

where *id* is the ID of the Space.

This returns a JSON object in the *data* portion of a successful JSON response of the form

```
{
  "id": id,
  "name": name,
  "description": description,
  "metadata", metadata,
  "liveActivityGroups": liveActivityGroups,
  "subspaces", subspaces
}
```

The names and a description of their values is given below.

id	The ID of the Space
name	The name of the Space
description	The description of the Space
metadata	The metadata of the Space

The *liveActivityGroups* field will be a list of information for each Live Activity Group in the Space. Each list element will have the form

```
{
  "id": groupId,
  "name": groupName,
  "description": groupDescription,
  "metadata", groupMetadata
}
```

The names and a description of their values is given below.

groupId	The ID of the Group
groupName	The name of the Group
groupDescription	The description of the Group
groupMetadata	The metadata of the Group

The *subspaces* field will be a list of information for each child Space in the Space. Each list element will have the form

```
{
  "id": subspaceId,
  "name": subspaceName,
  "description": subspaceDescription,
  "metadata", subspaceMetadata
}
```

The names and a description of their values is given below.

subspaceId	The ID of the child Space
subspaceName	The name of the child Space
subspaceDescription	The description of the child Space
subspaceMetadata	The metadata of the child Space

23.6.3 Requesting the Status of all Live Activities in a Space

A request to get a status update of all Live Activities in a Space can be initiated by the API command

```
/space/id/liveactivitystatus.json
```

where *id* is the ID of the Space.

The Live Activities in the Space is the set of all Live Activities in all Live Activity Groups in the space and all subspaces of the Space, and their subspaces.

The JSON returned will the simple JSON success result.

23.6.4 Deploying a Space

A Space is deployed by the API command

```
/space/id/deploy.json
```

where *id* is the ID of the Space.

Deploying a Space ultimately deploys all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.6.5 Configuring a Space

A Space is configured by the API command

`/space/id/configure.json`

where *id* is the ID of the Space.

Configuring a Space ultimately configures all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.6.6 Starting Up a Space

A Space is started up by the API command

`/space/id/startup.json`

where *id* is the ID of the Space.

Starting up a Space ultimately starts up all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.6.7 Activating a Space

A Space is activated by the API command

`/space/id/activate.json`

where *id* is the ID of the Space.

Activating a Space ultimately activates all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.6.8 Deactivating a Space

A Space is deactivated by calling the API command

`/space/id/deactivate.json`

where *id* is the ID of the Space.

Deactivating a Space ultimately deactivates all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.6.9 Shutting Down a Space

A Space is shut down calling the API command

```
/space/id/shutdown.json
```

where *id* is the ID of the Space.

Shutting down a Space ultimately shuts down all Live Activities defined in all Live Activity Groups in the Space and all child Spaces.

The JSON returned will the simple JSON success result.

23.7 Named Scripts

The Named Scripts API allows you to get a collection of all scripts known by the Master. Once you have the IDs, you can then run the scripts.

23.7.1 Getting All Named Scripts

The API call to get the list of all Named Scripts is

```
/admin/namedscript/all.json
```

This returns a JSON list in the *data* portion of a successful JSON response of the following form

```
{
  "id": id,
  "name": name,
  "description": description,
}
```

The names and a description of their values is given below.

id	The ID of the Script
name	The name of the Script
description	The description of the Script

23.7.2 Running a Named Script

A Named Script is run by the API command

```
/admin/namedscript/id/run.json
```

where *id* is the ID of the Script.

The JSON returned will the simple JSON success result.

CHAPTER

TWENTYFOUR

COOKBOOK

The following cookbook will point out various useful recipes for things you might want to do with Interactive Spaces.

24.1 Repeating An Action Over and Over

Sometimes you need to repeat some action over and over again. It is a bad idea to never exit any of your Live Activity's lifecycle methods, so another way is needed instead of a loop in a lifecycle method which never exits.

Interactive Spaces includes thread pools, which give you the ability to run multiple processes at the same time. In fact, you should never create a thread on your own, you should only use Interactive Space's thread pools as you can prevent a controller from being able to be easily shut down if you create your own threads.

The following examples show how to use the thread pools in a variety of languages. The task will be a simple one, print *Hello World* every 5 seconds. The very first time it will happen will be 10 seconds in the future.

The examples start up the thread when the Activity is activated. It will automatically be shutdown when the activity is shut down.

24.1.1 Python

```
class RepeatingActionExampleActivity(BaseActivity):
    def onActivityActivate(self):
        class Foo(Runnable):
            def run(self):
                print "Hello, world"

        self.managedCommands.scheduleWithFixedDelay(Foo(), 10, 5, TimeUnit.SECONDS)
```

24.2 Cloning A Space

A complex installation may sometimes have repetitive elements in it, where those elements are somewhat complex themselves. For example, there may be a Space with child Spaces, many of them with Live Activity Groups containing Live Activities which need to run on different Space Controllers. Duplicating these complex networks manually could be quite timeconsuming.

This cloning operation can be simplified by cloning the Space. This will in turn clone all the child objects.

You set the *namePrefix* to be a name which will be prefixed to all generated objects.

You can map how you want the Space Controllers in the original Space to map into the Space Controllers in cloned Live Activities by using a *controllerMap*. If there is no map supplied, or no mapping found for a particular Space Controller in the *controllerMap*, the Space Controllers used in the clone will be the same as the Space Controller in the original.

The following examples show how to clone a space in a variety of scripting languages. They will prepend *Clone Test* to all cloned objects and map only 1 Space Controller to another Space Controller.

24.2.1 Groovy

```
def cloner = new interactivespaces.master.server.services.SpaceCloner(activityRepository, spaceReposi
cloner.namePrefix = 'Clone Test';

def controller1 = controllerRepository.getSpaceControllerByUuid('8826e47c-a08a-4b3c-a320-06f420a33904
def controller2 = controllerRepository.getSpaceControllerByUuid('1a32c84a-3786-4329-9ae3-31c9424823d5
cloner.controllerMap = [(controller1): controller2];

def space = spaceRepository.getSpaceById('cfdce17a-d841-485a-9214-e2e47e4865a6');
cloner.cloneSpace(space);
cloner.saveClones();
```