IOLITE APP DEVELOPMENT GUIDE

SPECIFICATION

Last modification:	08.03.2016
Status:	Draft
Confidentiality:	Confidential, do not redistribute
Main contact:	Grzegorz Lehmann (CTO) grzegorz.lehmann@iolite.de
Owner:	IOLITE GmbH Helmholtzstr. 2-9, 10587 Berlin Handelsregister: Amtsgericht Charlottenburg HRB 169900 B Geschäftsführer: Grzegorz Lehmann

Version	Author	Modifications
02.06.2015	Grzegorz Lehmann (IOLITE)	• Initial version
28.09.2015	Grzegorz Lehmann (IOLITE)	• Update with time series access
04.11.2015	Grzegorz Lehmann (IOLITE)	• Add Maven build information
05.11.2015	Grzegorz Lehmann (IOLITE)	• Fix App descriptor file name • Add logging information
11.11.2015	Grzegorz Lehmann (IOLITE)	• Set maven-assembly-plugin version to 2.6 • Add DeviceAPI package to example to clarify the correct import
19.11.2015	Grzegorz Lehmann (IOLITE)	• Update Maven repository URL • Updated historical data examples • Add info about JavaDoc with Maven
18.12.2015	Grzegorz Lehmann (IOLITE) Falko Schwabe (IOLITE)	• Document JavaScript access to app APIs • Document session token handling for UIs
11.02.2016	Grzegorz Lehmann (IOLITE)	• Document User API and Heating API
08.03.2016	Grzegorz Lehmann (IOLITE)	• New Eclipse update site URL

CONTENTS

INTRODUCTION

This document provides a guide to app development for the IOLITE platform. The target audiences are programmers interested in deploying new applications into the IOLITE ecosystem.

The idea of IOLITE is that any developer should be enabled and empowered to build IOLITE Apps so that the ecosystem grows and the users gain access to new, innovative and useful functions. IOLITE Apps are built in order to provide new functionality to the smart home of the user. As such, they are extensions of the IOLITE system, implementing smart home use cases based on the APIs provided by the IOLITE Runtime.

REQUIREMENTS

For the IOLITE App and IOLITE Driver development you need:

- Java SE 1.6 or higher
- Eclipse IDE with Eclipse Modeling Framework (EMF) and Plugin Development Environment (PDE). The easiest way to obtain the two is to download the Eclipse Modeling Tools edition of Eclipse (you can find it in the list here http://www.eclipse.org/downloads/).
- IOLITE SDK Eclipse Plugins, which can be installed from the IOLITE Update Site (https://dev.iolite.de/update-site/)

BASICS

An IOLITE App is packaged as a Java Archive (JAR) and consists of the following parts:

- IOLITE App XML descriptor – holding basic meta-information
- IOLITE App Java code – application logic executed within a sandbox provided by the IOLITE Runtime
- IOLITE UI code – user interface part of the app, implemented in JavaScript/HTML

The IOLITE Runtime provides the apps with a set of s.c. IOLITE App APIs, through which the apps gain access to the smart home devices & sensors, user data, etc. Following IOLITE App APIs are available at this moment:

- Device API - provides access to device & sensor data as well as the possibility of executing actions
- Storage API - provides a persistent storage for the data of the app
- Frontend API - enables the app to expose a user interface via HTTP
- Environment API - provides information about the smart home environment in terms of its locations and their elements

The above listed IOLITE App API set will be extended in the future.

The apps can access the APIs through both Java (POJO) and JavaScript (JSON/Websocket).

APP RESTRICTIONS

IOLITE Apps are not allowed to (while not implemented at this moment, this will be enforced by the Runtime in short future):

- Open network sockets or server sockets. If an app needs access to the network, it has to request permission for the Network API. Networking functions will then be provided via the Network API.
- Access the file system in any way. In an app needs access to persistent storage, it must use the Storage API.
- Apps should not start own threads. The provided `Scheduler` should be used.

Further, IOLITE Apps should be Java 1.6 compatible, since the IOLITE Runtime can deployed to hardware gateways that do not provide higher Java versions.

APP EXECUTION RULES

Please note the following rules of IOLITE App execution:

1. An IOLITE App is always installed for a user. If the app has a UI, only this user can access it. If other users wish to access the app, they have to install it for themselves.
2. An IOLITE App installed for a user is called an IOLITE App Instance (IOLITE App + IOLITE User = IOLITE App Instance).
3. Each IOLITE App Instance is executed for itself. The instances cannot communicate with each other (e.g. to not compromise potentially private data between users). Please keep this in mind when designing your app.
4. Each IOLITE App Instance is started immediately when the IOLITE Runtime starts (or, for newly installed applications, right after the application is installed).
5. If the IOLITE App features a user interface (equivalent to using the Frontend API), it will be displayed in the Home Control Center (HCC) user interface in the Apps area.
6. Users can only access the UI of their app instances.

APP MODULE

The IOLITE App must have one main app class extending `de.iolite.app.AbstractIOLITEApp`. This is the interface between the app and the IOLITE runtime.

A project module for an IOLITE App typically features the following classpath elements:

1. The main class of the app extending `AbstractIOLITEApp` and all other classes as well as resources required by the app.
2. IOLITE libraries (IOLITE App API interfaces and accompanying modules used by the app classes)
3. App libraries

Each IOLITE App is packaged as one JAR file. The JAR of the app must include classpath elements 1. and 3. from the above list. The IOLITE libraries should not be packaged into the JAR.

IOLITE does not impose any particular project style for building the JAR. However, support for Eclipse Plugin projects and Maven projects are provided.

APP AS ECLIPSE PLUGIN

If you set up your IOLITE App module as an Eclipse plugin, you need to add the following plugin dependencies into the `MANIFEST.MF`:

```
Require-Bundle: org.eclipse.emf.ecore;bundle-version="2.10.0",
 de.iolite.app.api.app-api-common;bundle-version="1.0.0",
 de.iolite.common.lifecycle;bundle-version="1.0.0",
 de.iolite.app.api.device-api.access;bundle-version="1.0.0",
 de.iolite.runtime.api.runtime-api-common;bundle-version="1.0.0",
 de.iolite.app.api.device-api;bundle-version="1.0.0",
 de.iolite.app.api.frontend-api;bundle-version="1.0.0",
 de.iolite.app.api.environment-api;bundle-version="1.0.0",
 de.iolite.app.api.storage-api;bundle-version="1.0.0",
 de.iolite.utilities.time-series;bundle-version="1.0.0",
 de.iolite.utilities.concurrency-utils;bundle-version="1.0.0",
 de.iolite.drivers.iolite-driver-api;bundle-version="1.0.0",
 de.iolite.utilities.disposeable;bundle-version="1.0.0",
 de.iolite.common.entity-identifier;bundle-version="1.0.0",
 org.slf4j.slf4j-api;bundle-version="1.7.7",
 de.iolite.app.api.heating-api.access;bundle-version="1.0.0",
 de.iolite.app.api.user-api.access;bundle-version="1.0.0"
```

APP AS MAVEN[1] PROJECT

To access IOLITE Maven repositories, an account is needed. Please contact the IOLITE team to setup an account.

All needed IOLITE modules are hosted in one repository. Please add the following repository configuration to your Maven `settings.xml` (or your project configuration, depending on your preference):

[1] https://maven.apache.org

```xml
<repository>
    <id>iolite-snapshots</id>
    <name>IOLITE API Snapshots</name>
    <url>https://web.iolite.de/nexus/content/repositories/iolite-
snapshots/</url>
    <releases>
        <enabled>false</enabled>
    </releases>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
</repository>
```

The app module needs the following dependencies:

```xml
<!-- IOLITE App API Common -->
<dependency>
    <groupId>de.iolite.app.api</groupId>
    <artifactId>app-api-common</artifactId>
    <version>0.1-SNAPSHOT</version>
    <!-- set to provided to exclude the dependency from final JAR -->
    <scope>provided</scope>
</dependency>
<!-- IOLITE App APIs -->
<dependency>
    <groupId>de.iolite.app.api</groupId>
    <artifactId>app-apis</artifactId>
    <version>0.1-SNAPSHOT</version>
    <!-- set to provided to exclude the dependency from final JAR -->
    <scope>provided</scope>
</dependency>
```

When adding IOLITE dependencies, please make sure to set the scope to `provided` (as in the example above) to exclude the API JARs from the final JAR built by Maven.

To create the final JAR comfortably, the use of the `maven-assembly-plugin` is recommended. The following example configuration can be used in the `pom.xml`:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>

                <!-- Feel free to set a name here -->
                <finalName>${project.artifactId}</finalName>
                <appendAssemblyId>false</appendAssemblyId>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

LOGGING

IOLITE uses SLF4J[2] facade with Logback[3] for logging. The Maven dependencies are as follows:

```
<dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.7</version>
      <scope>provided</scope>
</dependency>
<dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.1.2</version>
      <scope>provided</scope>
</dependency>
```

To obtain the logger, please use the `org.slf4j.LoggerFactory` and your class as parameter. Example:

```
static final Logger LOGGER = LoggerFactory.getLogger(<Your Class>.class);
```

From now on, the logger can be used, for example:

```
LOGGER.debug("Application started");
```

The logging can be controlled by means of the `logback.xml` configuration. This file needs to be put into `<user-home>/.iolite` directory and IOLITE will load it automatically.

Please check the Logback website for details of the configuration[4].

APP DESCRIPTOR

In order to be deployed in the IOLITE Runtime, each app must be described in an `IOLITEApp.xml`.

The XML descriptor file must be placed in the root classpath container of the app module.

The app descriptor can be initialized automatically by the IOLITE SDK:

1. Right-click on a resource within the driver project
2. Go to IOLITE App sub-menu
3. Select Generate App Descriptor XML

[2] http://www.slf4j.org/
[3] http://logback.qos.ch/
[4] http://logback.qos.ch/manual/configuration.html

The SDK automatically puts the descriptor in one of the Java source folders and writes the name of your IOLITE App class into it.

APP DESCRIPTOR FORMAT

The App descriptor is an XML file specifying:

- meta-data about the app, e.g. its name
- name of the main app class to execute
- permissions that the app needs in order to work

APP DESCRIPTOR EXAMPLE

The following XML is an example IOLITE App descriptor. The app's main class is `de.iolite.app.example.ExampleApp`. The app requests several API permissions.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:iolite-app-configuration
      xmlns:ns2="www.iolite.de/iolite-app-configuration"
      name="Example App"
      className="de.iolite.app.example.ExampleApp">

      <api-permission
            name="de.iolite.app.api.frontend.FrontendAPI"
            type="READ_WRITE"/>
      <api-permission
            name="de.iolite.app.api.storage.StorageAPI"
            type="READ_WRITE"/>
      <api-permission
            name="de.iolite.app.api.device.access.DeviceAPI"
            type="READ_WRITE"/>
      <api-permission
            name="de.iolite.app.api.environment.EnvironmentAPI"
            type="READ_WRITE"/>
      <api-permission
            name=" de.iolite.api.heating.access.HeatingAPI"
            type="READ_WRITE"/>
      <api-permission
            name="de.iolite.app.api.user.access.UserAPI"
            type="READ_WRITE"/>

      <icon-url>de/iolite/app/example/ioliteicon.png</icon-url>

      <preferred-width>0</preferred-width>
      <preferred-height>0</preferred-height>

</ns2:iolite-app-configuration>
```

JAVADOC

The IOLITE Maven repository holds JavaDoc JARs for each module. You can use Maven to retrieve those. For example, if you setup your project for Eclipse, the `mvn-eclipse-plugin` can download the javadocs:

```
mvn eclipse:eclipse -DdownloadJavadocs=true
```

APP IMPLEMENTATION

Each IOLITE App features a main class, which is loaded by IOLITE after the app's installation. The main app class must extend the abstract `de.iolite.app.AbstractIOLITEApp` class and implement the following methods:

- `initializeHook()` – called by IOLITE during initialization of the app
- `startHook(IOLITEAPIProvider context)` – called by IOLITE when the app is supposed to start its functionality. The `IOLITEAPIProvider` parameter provides the app with access to IOLITE APIs.
- `stopHook()` – stops the app. The app should stop all activity when this method is called.
- `cleanUpHook()` – called after stop, the app should free its resources.

From the above methods, `startHook` is usually the most important one. Here the app retrieves the IOLITE APIs and starts its activities.

ACCESSING IOLITE APIS

The `IOLITEAPIProvider` provided in `startHook` gives the app access to IOLITE APIs. An API instance is retrieved by means of the `getAPI(Class apiClass)` method, for example:

```
DeviceAPI deviceAPI =
context.getAPI(de.iolite.app.api.device.access.DeviceAPI.class);
```

For each API, the particular API interface class needs to be used.

SCHEDULER

IOLITE apps must not start their own threads. In order to lunch recurring or parallel tasks, a scheduler interface is provided by the `IOLITEAPIProvider`:

```
Scheduler scheduler = context.getScheduler();
```

The scheduler offers methods for executing parallel tasks, scheduling task with a delay as well as recurring tasks.

Please note that IOLITE may assign a limited number of threads to each app instance (e.g. due to CPU limitations). Try to scheduler as little tasks as possible.

DEVICE API

The IOLITE DEVICE APP API provides IOLITE Apps access to the devices and sensors of the smart home environment. The API enables to read information about the devices and to control them. The access is highly restricted, based on the permissions of the app.

ACCESSING DEVICE API

IOLITE Apps that want to access the Device API need to set the following permission:

```
<api-permission
    name="de.iolite.app.api.device.access.DeviceAPI"
    type="READ_WRITE"/>
```

As for any IOLITE App API, the `IOLITEAPIProvider` of the app provides the API access:

```
final de.iolite.app.api.device.access.DeviceAPI deviceAPI =
getContext().getAPI(DeviceAPI.class);
```

DEVICE API DATA STRUCTURE

The `DeviceAPI` object holds all devices available to the app in the `devices` list. Each `Device` has the following attributes:

- `name` - user-friendly name of the device
- `identifier` - unique (throughout IOLITE) identifier of the device
- `manufacturer` - name of the manufacturer of the device
- `modelName` - model name of the device (optional)

Each device consists of properties. A property represents a read-only or read/write value of the device. The properties of the devices are held in the `Device.properties` list. Each property has a key (identifying the type of the property, matching the property type identifier used in the app permissions, e.g. `http://iolite.de#powerUsage or http://iolite.de#on`) and a `value` (holding the current value of the property).

Each device has a s.c. `PropertyProfile`. The profile can be seen as the type of the device. It defines of which properties the device consists. The profile of a `Device` is identified by the `profileIdentifier` attribute. Example `profileIdentifier` values are `http://iolite.de#Lamp`, `http://iolite.de#Oven`, `http://iolite.de#TemperatureSensor`.

The property type identified by the property key, defines the property meta-data, including unit, minimum and maximum values, step interval, etc. This meta data is exposed via the `DeviceAPI`:

- `getProfiles()` – returns all property profiles known to IOLITE
- `getProfile(String profileIdentifier)` – returns a property profile with a given identifier
- `getPropertyTypes()` – returns all property types known to IOLITE
- `getPropertyType(String key)` – returns a property type for a device property key

The history of each property is stored by IOLITE and can be retrieved by an app, assuming the app has at least read access to that property. The examples are provided below.

READING DEVICE INFORMATION

The following example code iterates through all devices and reads the value of the on/off status property (`http://iolite.de#on` property type stored in constants of `DriverConstants` utility):

```
// go through all devices
for (final Device device : deviceAPI.getDevices()) {
    // let's get the 'on/off' status property
    final DeviceBooleanProperty onProperty =
device.getBooleanProperty(DriverConstants.PROPERTY_on_ID);
    // check if the device has the property
    if (onProperty != null) {
        // property set, log state of device
        final boolean isDeviceOn = onProperty.getValue();
        LOGGER.debug(MessageFormat.format("Device ''{0}'' is {1}",
device.getIdentifier(), isDeviceOn ? "on" : "off"));
    }
}
```

CONTROLLING DEVICES

Device control works by changing the values of properties. For example, to turn a device on, value `true` has to be set in the `http://iolite.de#on` property. Similarly, to turn the device off, the `http://iolite.de#on` property has to be set to `false`.

Property values are changed by means of the `requestValueUpdate` operation.

```
void requestValueUpdate(String propertyValue) throws DeviceAPIException;
```

Please note that changing the value of a property may be a time-consuming process that continues even after the `requestValueUpdate` operation finished. Consider the example of window blinds - the process of driving out the blinds from 0% to 100% may take several seconds (depending on the size of the window or the blinds motor). Depending on the implementation of the device driver, the `requestValueUpdate` operation may or may not be blocking until the requested state is reached. Therefore, a successful execution of the `requestValueUpdate` operation does not imply that the value has been reached. To determine if the value has been set, please observe the `value` of the modified property.

The previous example can be extended in the way that the device on/off status is changed:

```
// go through all devices
for (final Device device : deviceAPI.getDevices()) {
    // let's get the 'on/off' status property
    final DeviceBooleanProperty onProperty =
device.getBooleanProperty(DriverConstants.PROPERTY_on_ID);
    // check if the device has the property
    if (onProperty != null) {
        // property set, log state of device
        final boolean isDeviceOn = onProperty.getValue();
        LOGGER.debug(MessageFormat.format("Device ''{0}'' is {1}",
device.getIdentifier(), isDeviceOn ? "on" : "off"));
        // invert the state (if device is on, turn it off and vice versa)
```

```
            onProperty.requestValueUpdate(!isDeviceOn);
        }
}
```

OBSERVING DEVICES

The `DeviceAPI` provides an observer pattern implementation. Each element of the API can be observed by means of an `setObserver(..)` method. In the following the running example is extended so that the value of the on/off property is observed.

```
// observe value of property
onProperty.setObserver(new DeviceBooleanPropertyObserver () {

    /**
     * {@inheritDoc}
     */
    @Override
    public void valueChanged(final Boolean value) {
        if (value) {
            LOGGER.debug("Device turned on");
        }
        else {
            LOGGER.debug("Device turned off");
        }
    }
});
```

READING DEVICE HISTORY

IOLITE automatically stores device property values in a database. The Device API provides access to the historical values for each property:

- `getValuesSince(long start)` – returns the historical values of a property from the `start` timestamp until present. Please note that for performance reasons the IOLITE runtime may limit the number of returned entries. The default limit is 250, but it may vary based e.g. on the computing power of the platform. Please make sure to check the timestamp of the newest entry.
- `getAggregatedValuesSince(long start, TimeInterval resolution, Function function)` – returns the historical values of a property from the `start` timestamp until present, with a given resolution and aggregation function. Please note that if no data entries are available for a resolution interval, a `NaN` entry is returned for that interval.
- `getValuesBetween(long start, long end)` – returns a time series of historical data of a property between a `start` and `end` timestamp. Please note that for performance reasons the IOLITE runtime may limit the number of returned entries. The default limit is 250, but it may vary based e.g. on the computing power of the platform. Please make sure to check the timestamp of the newest entry.
- `getAggregatedValuesBetween(long start, long end, TimeInterval resolution, Function function)` – returns a time series of historical data of a property between a `start` and `end` timestamp, with a given resolution and aggregation function. Please note that if no data entries are available for a resolution interval, a `NaN` entry is returned for that interval.
- `getValuesOf(long date, TimeInterval timeWindow)` – returns a time series of historical data of a given time window falling into a given date. For example, if `timeWindow` is `Hour`, this operation returns only values of the hour into which date falls, e.g. if the `date` identifies a timestamp of 10:45am values between 10am and 11am will be returned. Please note that for performance reasons the IOLITE runtime may limit the number of returned entries. The

default limit is 250, but it may vary based e.g. on the computing power of the platform. Please make sure to check the timestamp of the newest entry.

- `getAggregatedValuesOf(long date, TimeInterval timeWindow, TimeInterval resolution, Function function)` – returns a time series of historical data of a given time window falling into a given date, with a given resolution and aggregation function. Please note that if no data entries are available for a resolution interval, a `NaN` entry is returned for that interval.

All the mentioned `getValues`… methods return results of type `java.util.List`. The elements of the list are descendants of `de.iolite.utilities.time.series.DataEntry` class. Depending on the property type, list elements can be of class `BooleanEntry`, `DoubleEntry`, `IntEntry`, `LongEntry` or `StringEntry`. The mentioned `getAggregatedValues`… methods return results of type `java.util.List`. The elements of the list are objects of `de.iolite.utilities.time.series.AggregatedEntry` class.

For example, the code below lists all devices and then logs the history of power usage property.

```
// retrieve historical time series of the property's value
// all timestamps are in milliseconds since epoch UTC
final DeviceDoubleProperty powerUsage =
device.getDoubleProperty(DriverConstants.PROPERTY_powerUsage_ID);
// get hourly value’s of today
final List<AggregatedEntry> history =
powerUsage.getAggregatedValuesOf(System.currentTimeMillis(), TimeInterval.Day,
TimeInterval.Hour, Function.Average);
// iterate over the time series
for (final AggregatedEntry entry : history) {
     LOGGER.info("The device used an average of {} Watt at '{}'.",
entry.getAggregatedValue(), entry.getEndTime());
}
```

STORAGE API

The IOLITE Storage API enables to store their persistently (that means stored data is still available after restart of the app or IOLITE).

ACCESSING STORAGE API

IOLITE Apps that want to access the Storage API need to set the following permission:

```
<api-permission
      name="de.iolite.app.api.storage.StorageAPI"
      type="READ_WRITE"/>
```

As for any IOLITE App API, the `IOLITEAppAPIProvider` of the app provides the API access:

```
final StorageAPI storageAPI = getContext().getAPI(StorageAPI.class);
```

READING AND STORING DATA

The `StorageAPI` provides a persistent key/value storage.

```
// storage API enables the App to store data persistently
// whatever is stored via the storage API will also be available if the App is
restarted
final StorageAPI storageAPI = context.getAPI(StorageAPI.class);

// Storage API provides a key/value storage for different data types
// save an integer under the key 'test'
storageAPI.saveInt("test", 10);
// now let's store a string
storageAPI.saveString("some key", "some value");
// log the value of an entry, just to demonstrate
LOGGER.debug("loading 'test' from storage: {}",
Integer.valueOf(storageAPI.loadInt("test")));
```

Please note that the data access is limited to the app instance. Each app instance has its own data and cannot access data of other app instances. That means, even other instances of the same app (started for different users) will not be able to see each other's data.

The API also provides operations for working with lists of values, such as `saveIntList`, `loadIntList`, `addIntToList`, etc.

FRONTEND API

The IOLITE Frontend API allows applications to expose an HTML/JavaScript user interface and handle HTTP requests. Via the Frontend API, apps can expose both static and dynamic (public and non-public) resources. The Frontend API is the only way for an IOLITE App to expose a user interface. Any UI resources, whether static or dynamic (public or non-public) have to be registered via the Frontend API.

The user interfaces exposed by the apps via the Frontend API are automatically embedded into IOLITE's Home Control Center (HCC) web-app. Each user can see her/his installed apps and access their user interface via the HCC.

IOLITE automatically takes care of client authentication and authorization when processing requests to resources exposed via the Frontend API. It is guaranteed that the resources of an app are only accessed by the user who owns a corresponding app instance.

ACCESSING FRONTEND API

IOLITE Apps that want to access the Frontend API need to set the following permission:

```
<api-permission
     name="de.iolite.app.api.frontend.FrontendAPI"
     type="READ_WRITE"/>
```

As for any IOLITE App API, the `IOLITEAppAPIProvider` of the app provides the API access:

```
final FrontendAPI frontendAPI = getContext().getAPI(FrontendAPI.class);
```

REQUEST HANDLERS

The Frontend API uses s.c. request handlers (implementing the `de.iolite.common.requesthandler.IOLITEHTTPRequestHandler` interface) to expose user interface resources. Each app can register request handlers for paths, relative to the root URL address of the app. Whenever an authorized request is made to a path, the handler registered with this path is called. If no handler is registered for a given path, the s.c. default handler is called.

IOLITE provides a set of predefined request handlers (more on that in a moment). If the app needs to expose dynamic resources, it can implement own request handlers. The best way to do it is to extends the abstract `de.iolite.app.api.frontend.util.FrontendAPIRequestHandler` class. However, let us get to the simple examples first.

STATIC REQUEST HANDLERS

The Frontend API already provides request handlers for static resources, like images, static html files, etc. The following example registers a classpath resource `de/iolite/app/example/ioliteicon.png` under the relative path `ioliteicon.png`. In other words, whenever a request is made to `<root URL of app>/ioliteicon.png`, the image from the denoted classpath resource will be delivered.

```
// give access to private static resource under a specific relative path
```

```
frontendAPI.registerClasspathStaticResource("ioliteicon.png",
"de/iolite/app/example/ioliteicon.png");
// give access to public static resource under a specific relative path
frontendAPI.registerPublicClasspathStaticResource("ioliteicon.png",
"de/iolite/app/example/ioliteicon.png");
```

Sometimes the app wishes to have more control over the request handler (e.g. specify the content-type of the reponse). In such cases, the `ClasspathStaticRequestHandler` can be used:

```
// for private resources
frontendAPI.registerRequestHandler("index.html", new
ClasspathStaticRequestHandler(IOLITEHTTPResponse.HTML_CONTENT_TYPE,
"de/iolite/app/example/index.html", ExampleApp.class.getClassLoader()));
});
// ... similar for public resources
frontendAPI.registerPublicRequestHandler("about.html", new
ClasspathStaticRequestHandler(IOLITEHTTPResponse.HTML_CONTENT_TYPE,
    "de/iolite/app/example/about.html", ExampleApp.class.getClassLoader()));
});
```

Very often the app has a multitude of UI resources and it would be tedious to pass each of them explicitly to the Frontend API. In such cases, the `FrontendAPIUtility` and `StaticResources` utility classes can be used:

```
// for private resources
FrontendAPIUtility.registerHandlers(frontendAPI,
StaticResources.scanClasspath("de/iolite/app/example/images",
getClass().getClassLoader()));
// ... similar for public resources
FrontendAPIUtility.registerPublicHandlers(frontendAPI,
StaticResources.scanClasspath("de/iolite/app/example/images",
getClass().getClassLoader()));
```

In the above example, the `StaticResources` utility traverses all resources found in the denoted package (in the above example `de/iolite/app/example/images`) as well as its children. Any resource found is returned with a path relative to the root package. The returned resources can be passed to the `FrontendAPIUtility`, which registers them as static resources in the Frontend API.

DYNAMIC REQUEST HANDLERS

When the app exposes dynamic resources, it can implement own request handlers, by extending the `FrontendAPIRequestHandler` class. This gives the app the full control over the request and the generated response. In the following example a dynamic request handler is registered:

```
frontendAPI.registerRequestHandler("index.html", new
FrontendAPIRequestHandler() {
    /**
     * {@inheritDoc}
     */
    @Override
    protected IOLITEHTTPResponse handleRequest(final IOLITEHTTPRequest
request, final String subPath) {
        // this handler simply delivers some HTML content
        final StringBuilder output = new StringBuilder();
        output.append("<html><head>");
        output.append("<script src='/hcc/jquery-2.1.0.min.js'></script>");
        output.append("</head><body>");
        output.append("<p>Example IOLITE App is working!</p>");
        output.append("<img src='logo/IOLITE-LOGO_Complex.png'
alt='IOLITE'>");
        output.append("</body></html>");

        // deliver the output in a response object
        return new IOLITEHTTPStaticResponse(output.toString(),
IOLITEHTTPResponse.HTML_CONTENT_TYPE);
    }
});
```

The first argument for `registerRequestHandler` is the URL path, relative to the root URL of the app (i.e. `"index.html"`). The root URL of the app is generated by the IOLITE Runtime and passed to the HCC automatically. The app should not care for its root URL.

The second argument of `registerRequestHandler` is the actual handler of HTTP request. It has only one method - `handleReuqest`. That method is called when HTTP request arrives for the registered URL. Request itself is passed as a parameter (you can get request method, cookies, parameters, etc.). It is up to app developer to shape the response - set response content, type, HTTP status, etc. The provided `IOLITEHTTPStaticResponse` class should suffice for most usage scenarios.

You can also use `registerPublicRequestHandler` to register a request handler as a public resource. However, this type of handlers must only be used by the app to expose security-irrelevant resources (e.g. a logo or an icon). As a matter of fact it is possible that the usage of public request handlers in apps will be strongly limited in future IOLITE releases.

A special type of handler is the default handler. If a client makes a request to a path, for which there is no handler registered or which uses an invalid session token, the Frontend API will call the default handler. This handler shall not deal with such requests to provide a valid response as fallback but provide an error message like response for the client. For example, the default handler can respond with an error page:

```
frontendAPI.registerDefaultRequestHandler(new FrontendAPIRequestHandler() {

  protected IOLITEHTTPResponse handleRequest(final IOLITEHTTPRequest request,
final String subPath) {
    final StringBuilder output = new StringBuilder();
    output.append("<html><head></head><body>");
    output.append("<p>This is an error page</p>");
    output.append("</body></html>");
    // deliver the output in a response object
    return new IOLITEHTTPStaticResponse(output.toString(),
IOLITEHTTPResponse.HTML_CONTENT_TYPE);
  }
});
```

UNREGISTERING HANDLERS

Each handler can be unregistered using its relative path, e.g.: `unregister("test")`.

To unregister all the handlers, use `unregisterAll()`.

Please note that all handlers registered by an app are automatically unregistered when the app stops.

COMMUNICATION BETWEEN THE APP JAVA CODE AND THE APP UI

The main point of the Frontend API is to enable the HTML/JavaScript user interface of the app to safely communicate with the app's Java code, executed in the IOLITE runtime. IOLITE automatically wraps the handler mechanism with client authentication and authorization, implemented using session tokens. By embedding the app UI into the HCC, it is assured that only an authenticated and authorized user is able to access the resources exposed by the handlers of the app. Furthermore, IOLITE assures that the JavaScript of an app cannot access handlers of other apps.

The implementation of the session management has some implications for the JavaScript of your app. First of all, if you want to provide a user interface (UI) for your app, then registering request handlers, which respond to the client with HTML/JavaScript/Images, is your **only** way to go. Further, IOLITE provides your JavaScript with a session token, which you will need to embed into your requests to your app's Frontend API handlers.

The first step of the app UI integration is to make sure, the app is embedded into the Home Control Center (HCC) correctly.

EMBEDDING OF THE UI IN THE HCC UI

The HCC features the *IOLITE Apps* area, which enables access to the IOLITE App Stores[5], as well as already installed apps. As soon as an app is installed, it is made visible as an app item in the *Installed* tab, as depicted in Figure 1.

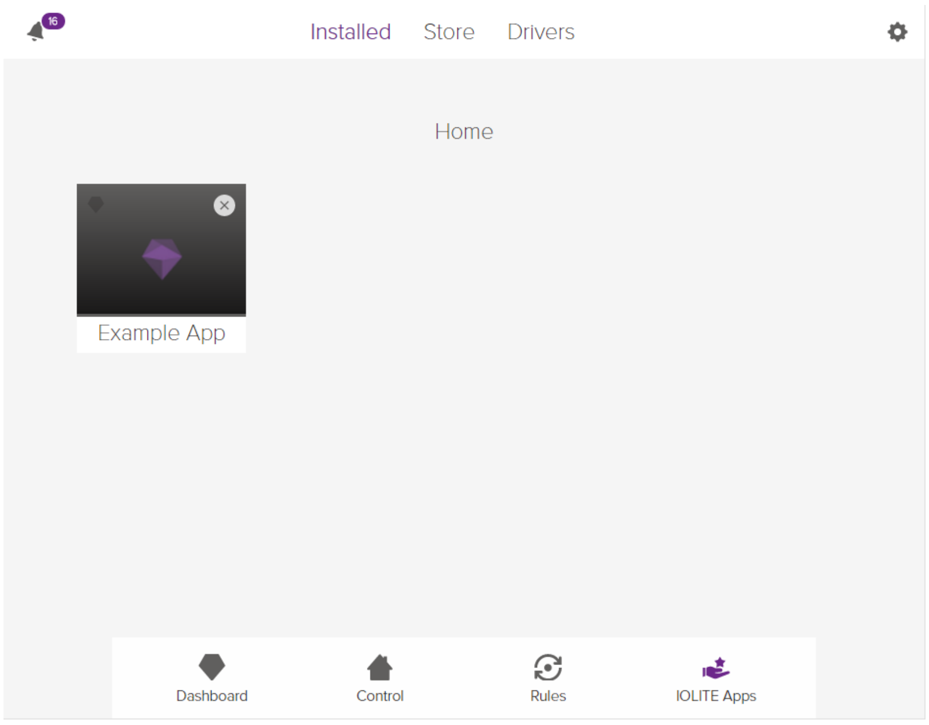
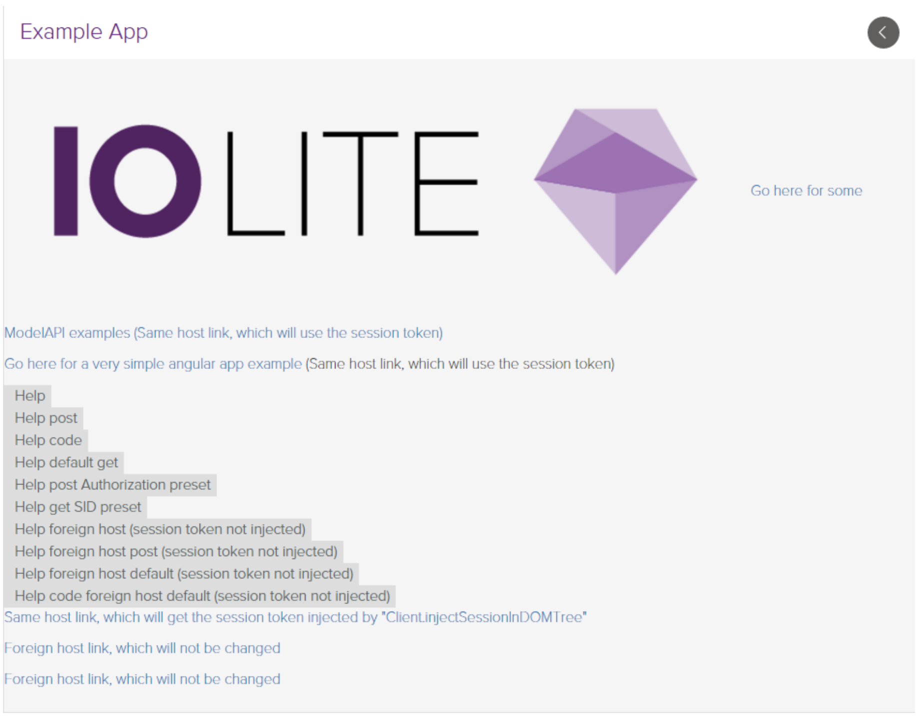


Figure 1: Installed app visualized in the Home Control Center

Clicking on the app icon of the app item opens your app UI in an embedded `<iframe>`. This `<iframe>` and the app title bar overlay of the HCC UI is displayed as long as the user stays within your app UI, as

[5] Please note that there are different kinds of apps stores. For example, there are online app stores but the `<home>/.iolite/apps` folder is treated as an app store as well. Thus, putting your app JAR into that directory makes it available in the Store area in the HCC.

shown in Figure 2. When the user uses the top-right located app-close button, the `<iframe>` and thus your app UI will be closed.



Go here for some

ModelAPI examples (Same host link, which will use the session token)

Go here for a very simple angular app example (Same host link, which will use the session token)

Help
Help post
Help code
Help default get
Help post Authorization preset
Help get SID preset
Help foreign host (session token not injected)
Help foreign host post (session token not injected)
Help foreign host default (session token not injected)
Help code foreign host default (session token not injected)

Same host link, which will get the session token injected by "Client.injectSessionInDOMTree"

Foreign host link, which will not be changed

Foreign host link, which will not be changed

Figure 2: "Example App" UI embedded into the HCC. Top bar is generated by IOLITE (with app name on the left and the '<' button on the right, bringing the user back to HCC).

Notably, the `<iframe>` is sandboxed and only allows the execution of JavaScript and the use of HTML forms[6]. Therefore, the access to the HCC UI's DOM tree, the `window.localStorage`, `window.sessionStorage` and `window.cookies` is prohibited by the browser, since the HCC UI does not set the sandbox-attribute `allow-same-origin`.

Furthermore, the so called origin of your app UI will not be the same as the one of the HCC UI. In fact the browser of the client will reset it to the string `null` instead. This requires adapting the AJAX requests (`XMLHttpRequests`) of the client as well as the request handlers of your app or otherwise they will fail, because the browser treats it as an invalid cross-origin access, even though you access technically the same origin as the HTML/JavaScript was delivered from. Therefore, please rely on so called `Access-Control XHR` requests[7]. In the code of your app, you can use the IOLITE request handler base class `de.iolite.common.requesthandler.AbstractHTTPRequestHandler`, which deals with the Access-Control handling and enables easy adaption of it.

Due to the sandboxing, session control in app UIs is handled differently based on the fact that the origin of the `<iframe>` is treated by the browser uniquely and independently from the origin of the HCC UI. Thus, IOLITE cannot safely rely on a cookie-based session control. Instead, we use `Java Web Tokens` formatted special request parameters. The following section describes this kind of session control and how you can use it for your app UI.

AUTHORIZATION AND AUTHENTICATION WITH SESSION TOKENS

[6] More on sandboxing: http://www.w3schools.com/tags/att_iframe_sandbox.asp
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

As described above, the Frontend API enables registration of public and non-public request handlers. Access to public request handlers does not require authorization and are thus accessible without providing a session token in the request. This type of handlers must only be used by the app to expose security irrelevant resources (e.g. HTML, JS, images or other static resources). Thus, do not use public request handlers, which have or need access to IOLITE App APIs or even to your own internal Java objects. As a matter of fact it is possible that the usage of public request handlers in apps will be **strongly limited** in future IOLITE releases. Please use private request handlers whenever possible.

Request to all non-public handlers, require a valid session token. There are two ways of adding the session token to your requests:

1. As part of the request header parameter, named `Authorization`. This is the **strongly preferred** solution.
2. As part of the request parameters in the URL or the content part of the request. An HTTP `GET` request needs to provide the session token in the URL parameter `SID`. An HTTP `POST` request must provide the very same as part of the request content instead.
   Please note that option 1 is more preferable due to security concerns. Having the session token integrated in the request URL implies that requests made by the HTML/JS in the client's browser will have the `referrer` set with the request URL. This includes the URL parameter `SID` with the session token. This is not a problem as long as your JavaScript only performs requests to the app's request handlers. However, accessing external resources (e.g. foreign webservers) from your app UI's HTML/JS will **leak the session token** to those (possibly malicious) servers.

For both options the value, i.e. the session token, is formatted as `Java Web Token` containing additional session-based information. It does not contain any more valuable information such as the plain-text passwords or other secret keys. The whole JWT session token represents the session assigned to the app instance of your app. It also has an expiry date, known only to the IOLITE server. The session time is extended whenever it is found to be used by an incoming request and when it is not expired yet. In other words – each request you make, prolongs your session.

The session token for the app UI is only issued, when the user decides to launch your app UI by clicking on the corresponding app item in the HCC. After that, restoring an expired session is not possible and can only be renewed by (closing and) reopening the app `<iframe>` in the HCC.

Due to the sandboxing of the app's `<iframe>`, the session token can only be given to the app UI in two possible ways. The very first request to the empty relative path (equals the empty string `""`) includes the issued session token as part of the URL in `SID`. This way, your request handler registered to this empty relative path will get to know the session token by querying it with `IOLITEHTTPRequest.getSessionToken()`. It then has two options:

- Return a response containing HTML dynamically composed in your request handler, which will contain the session token in some way. This method can also be seen as a way to deal with the previously mentioned option 1 of adding the session token to requests using the request header parameter `Authorization`.
- Return a response containing static HTML, which reads the session token from the URL via JavaScript. This method is compatible with the previously mentioned option 2 of adding the session token to requests using the URL parameter `SID`.

Recapping the described session control, only the very first request to the empty relative path is guaranteed to carry the session token. For any following request, your app is responsible for carrying on the session token. Since the session control is not cookie-based, your app UI requests won't carry the token automatically. However, IOLITE provides some helper JavaScript to ease the session control for the app UIs.

JAVASCRIPT HELPER LIBRARIES

IOLITE provides your app with two helpers to connect the app UI to IOLITE. The first is the s.c. `ModelAPI` and is a websocket JSON-based access to the App API instances, to which your app's Java code has. The second helper is a s.c. JavaScript client bootstrapper, which helps the app UI dealing with the session token mechanism and setting up the `ModelAPI` JavaScript objects.

Note: the `ModelAPI` is described in detail in later sections. This section deals only with the JavaScript bootstrapping / setup; the actual usage instructions for the `ModelAPI` JavaScript are provided in the "JavaScript App API Access" section.

In order to use the helpers, you will need to first acquire the client-side JavaScript files. Just insert the following code into the `<head>` or `<body>` tag of your app UIs HTML:

```
<script type="text/javascript">
  window["Client"] = {
    SID : null, // set session token if required to be embedded rather than set via
URL!
    throwErrorOnBootFailure : true,
    readyCallback : function() {
      // Initiate stuff here, which depends on AppAPIs!
    },
    failureCallback : function() {
      // Do something when client initialization fails
    }
  };
</script>
<script type="text/javascript" src="/hcc/jquery-2.1.0.min.js"></script>
<script type="text/javascript" src="/bus/ModelAPICommunicationObject.js"></script>
<script type="text/javascript" src="/bus/ModelAPI.js"></script>
<script type="text/javascript"
src="/bus/ModelAPIWebsocketClientFactory.js"></script>
<script type="text/javascript" src="/bus/bus-client-bootstrap.js"></script>
```

The first `<script>` block configures the so called `Client` object before the connection to the server is set up. The second `<script>` block loads a recent `jQuery` 2.x version, which can currently be anything from 1.x to more recent versions. It is the same `jQuery` JavaScript as used within the HCC UI. The remaining `<script>` blocks load mandatory JavaScript, which provides the `ModelAPI` interfaces. The very last JavaScript will set up the `ModelAPI` by querying the `ModelAPIProfiles` configuration for your app from the IOLITE runtime.

Depending on how you have setup the `Client` object, it first checks the presence of the session token in its `Client.SID` property. If it is not set, the initialization tries to read the session token from the browser's `window.localStorage`. Since any app UI is in a sandboxed `<iframe>`, reading from it will fail (this trial is only of interest for the HCC but not for apps). However, the next trial is to read the session token from the `SID` URL parameter from `window.location`. If this fails too and thus a session token could not be acquired, the initialization will throw or at least log an error depending on what `throwErrorOnBootFailure` is set to.

After trying to get the session token set in the `Client` object, the initialization will query the `ModelAPIProfiles` configuration. This query requires a session token to be set, which also identifies with which configuration must be responded with. This configuration contains the websocket endpoints, which allow access to the App APIs. As soon as the `ModelAPIProfiles` configuration is acquired, the initialization is done and calls your provided `readyCallback`. Please note that only at this point your app UI code will be able to use the `ModelAPI` to access the App APIs and not earlier. There is also no need to manage own websocket connections as this is transparently managed by the `ModelAPI` with no more configuration afford.

If initialization fails for some reason, there are multiple ways of how errors are handled. In case of failing to acquire the session token, either an error is thrown or logged to the JS console. If retrieving the `ModelAPIProfiles` configuration fails, the `Client.failureCallback` provided by your app UI is called. Obviously, if the browser fails to load one of the mandatory JavaScript, nothing really useful happens. However, you can provide specific error handler code at the `<script>` elements by hand, for example:

```
<script type="text/javascript">
function handleScriptLoadError(param){alert(param);}
</script>
<script type="text/javascript" src="..." onerror="handleScriptLoadError('bla')"
onload="alert('good!')"></script>
```

Regarding the helper for the ease of use of the session token mechanism, you have multiple options:

- Provide the session token in the `Client` configuration before initialization. This is part of embedding the session token in the HTML response as described before and it is the only way to inject the session token for the initialization routine.
- Use the session token by taking it from `Client.SID` again after initialization.
- Use the helper method `Client.injectSessionInHREF(href)` to get a modified URL, which includes the session token in the `SID` URL parameter if and only if the URL satisfies having the same host address as the HTML was retrieved from. For any other `href`, which references another host or even another port, the session token will not be included.
- Use the helper method `Client.injectSessionInLink(linkElement)` to modify the `href`-URL of the `linkElement` (of type <a> or the `jQuery`-object pointing to it) to include the session token.
- Use the helper method `Client.injectSessionInForm(formElement)` which adds a hidden `<input>` field with name `SID` to embed the session token as a request parameter in the data of the `formElement` (of type <form> or the `jQuery`-object pointing to it) at submit.
- Use the helper method `Client.injectSessionInDOMTree()` to let all <a> and <form> elements get modified all at once.
- Use the helper method `Client.navigateWithSession(href)` to change `window.location` to the modified URL including the session token as it would be returned by `Client.injectSessionInHREF(href)`.

SIDE NOTES FOR USING ANGULAR

The initialization code will additionally provide the so called `busClientBootstrap` angular module, which injects mostly the same `Client` object's helper methods for using the session token wherever needed into the `$rootScope` of your angular app.

It might be convenient to use the app API access (provided by the `ModelAPI` JavaScript) as an angular service, which is provided by another optional JavaScript:

```
<script type="text/javascript" src="/hcc/modules/ModelAPIService.js"></script>
```

There are some **issues regarding using AngularJS** in the sandboxed `<iframe>`: The http-service of AngularJS is implemented to always send `XHRF` tags with a request. However, it tries to remember these tags for later response validation in cookies. Since accessing cookies from JavaScript will fail within the sandbox, the http-service is more or less unusable.

Because of the issues with the http-service, AngularJS cannot lazy-load templates. Your app will need to preload them into the template cache f.e. via `jQuery.ajax`. This will require telling AngularJS to start in two phases whereby the start is to be resumed into the second phase manually. AngularJS uses the two phase start if in JavaScript `window.name` is set to the string `'NG_DEFER_BOOTSTRAP!'` in before the AngularJS script is loaded.

USER API

The IOLITE USER APP API provides IOLITE Apps access to information about the App’s user.

ACCESSING DEVICE API

IOLITE Apps that want to access the User API need to set the following permission:

```
<api-permission
     name="de.iolite.app.api.user.access.UserAPI"
     type="READ_WRITE"/>
```

As for any IOLITE App API, the `IOLITEAPIProvider` of the app provides the API access:

```
final de.iolite.app.api.user.access.UserAPI deviceAPI =
getContext().getAPI(UserAPI.class);
```

READING USER INFORMATION

The User API provides a `User` object with information about the user (e.g. the unique identifier of the user).

```
final UserAPI api = context.getAPI(UserAPI.class);
LOGGER.debug("Running for user '{}'", api.getUser().getIdentifier());
```

JAVASCRIPT APP API ACCESS

In order to enable an easy implementation of feature-rich user interfaces, IOLITE's app APIs accessible on Java level are also accessible via a JSON/Websocket interface from JavaScript. Any data and operations available in the Java APIs are available in JavaScript. This section provides an introduction into the JavaScript libraries and the general usage patterns of the JSON/Websocket APIs. It starts off with a short example of accessing the Device API.

JAVASCRIPT DEVICE API ACCESS

In order to access and use the Device API in JS we need to use a tool called *Model API*, which will be explained in depth in a later part of the/another tutorial. Briefly summarized, it enables to query information from App APIs like the Device API, execute methods and to observe changes of query information.

Similar to the `IOLITEAPIProvider` in Java, there is the `ModelAPIProfiles` prototype in JS, which returns so called `ModelAPI` objects. Each `ModelAPI` object on the JavaScript client side represents an App API on the IOLITE server-side. Due to JavaScript and browser technology properties and restrictions, these tools are designed to work deferred. In other words, they require the specification of callback functions, which deal with so called `success` and `error` signaling cases. Thus, retrieving the `ModelAPI` object for the Device API looks like this:

```
ModelAPIProfiles.get(ModelAPIProfiles.deviceId, {
  success:function(deviceAPI) {
    console.log('DeviceAPI is ready');
    // Work with the deviceAPI here
  }
});
```

The identifier `ModelAPIProfiles.deviceId` is provided by IOLITE automatically and only exists, if the app has the permissions to access the Device API.

READING DEVICE INFORMATION

Now having the `ModelAPI` object `deviceAPI`, we are able to e.g. query information from the Device API. In the first step we will retrieve all devices by executing the `query` function:

```
deviceAPI.query({
  request : new QueryRequest(null, null, "devices"),
  success : function(devices) {

  }
});
```

Having all devices, we will now iterate through them and perform another query, this time for the `on` property (more specifically, the `on` property is identified with the URI `http://iolite.de#on`, so we will query all properties with this key). If a device has an `on` property, we will log its `boolean` value to the console:

```
devices.forEach(function(device){
  var onPropertyQuery = "devices[identifier='" + device.identifier +
"']/properties[key='http://iolite.de#on']";
  deviceAPI.query({
    request : new QueryRequest(null, null, onPropertyQuery),
    success : function(onProperties) {
      if(onProperties.length <= 0) {
        // Query was successful, but no on-property was found
        // for device
        return;
      }
      var onProperty = onProperties[0];
      console.info('Device ', device.identifier, ' is ', onProperty.value ? 'on' :
'off');
    }
  });
});
```

Please note that the query string (in the above example held in the `onPropertyQuery` variable) is constructed as an XPath[8] query. On server side, IOLITE evaluates the queries on the Java object structure of the API. Further, the returned JSON objects are 1:1 mappings of the Java objects on IOLITE side. Feel free to experiment with the XPath syntax and try out different queries.

In the first example above, the query string was simply `"devices"`. This is because on Java level, the Device API's root object provides a list of devices via the `getDevices()` operation. The result of the query is a list of devices, which are the JSON representation of the `Device` objects returned by the getter on Java level.

In the constructed `onPropertyQuery` you can see the usage of `device.identifier`. This is possible, because `de.iolite.app.api.device.access.Device` defines an `identifier` attribute.

Assuming the device identifier is `"xyz123"`, the resulting query for the `on` property is:

```
"devices[identifier='xyz123']/properties[key='http://iolite.de#on']"
```

The query filters the device elements in the `devices` list by their `identifier` (using the `[]` XPath operator) and of those, retrieves the `properties` list, which then again is filtered for properties only with the `key` `http://iolite.de#on`.

The generic query engine applied to IOLITE's APIs gives you big flexibility in retrieving API information for your user interface. The next example deals with action execution.

CONTROLLING DEVICES VIA JAVASCRIPT

In order to change a value of a device property on Java level, the `requestValueUpdate` method is invoked. In the `ModelAPI` JavaScript the method invocations are represented by s.c. `ActionRequests`. In the following we modify the previous Device API example by changing the value of the retrieved `on` properties:

```
deviceAPI.query({
  request : new QueryRequest(null, null, "devices"),
  success : function(devices) {
    devices.forEach(function(device){
      // instead of query, we require an action now
      var onPropertyQuery = "devices[identifier='" + device.identifier +
"']/properties[key='http://iolite.de#on']";
      deviceAPI.action({
        request : new ActionRequest(null, null, onPropertyQuery,
"requestValueUpdate", [ new ValueParameter(true) ]),
        success : function() {
```

[8] http://www.w3schools.com/xsl/xpath_syntax.asp

```
          console.info('Successfully executed requestValueUpdate');
        },
        error : function(deviceAPI, responseRequestID, responseErrorCode,
responseError) {
          console.error("Requesting value update of ", device, " failed due to ",
responseErrorCode, ": ", responseError);
        }
      });
    });
  }
});
```

The `ActionRequest` object uses the `onPropertyQuery` to identify the object upon which the action will be executed. The next parameter is the name of the method to invoke (in this case `requestValueUpdate`), followed by a list of arguments that should be passed to the method. The `requestValueUpdate` method expects only one argument and here we pass a `true` value. The constructed `ActionRequest` JavaScript object is equivalent to calling `requestValueUpdate("true")` for the `on` property of the device in Java.

Please note that two types of parameters can be used in action requests. The `ValueParameter` holds a fixed value, whereas a `QueryParameter` holds a query to the object that should be used as parameter. When the action is executed, IOLITE evaluates the `QueryParameter` and uses the result of the query as the method call argument (more on this in later sections).

RETRIEVING HISTORICAL DEVICE DATA IN JAVASCRIPT

Another example for action execution is the retrieval of historical device data. In Java the Device API provides the historical device values via the `getValuesOf` methods. Thus, in JavaScript we need to invoke the `getValuesOf` method of a device property by means of an `ActionRequest`.

Below we retrieve the historical values of the `on` property of the queried devices:

```
deviceAPI.query({
  request : new QueryRequest(null, null, "devices"),
  success : function(devices) {
    devices.forEach(function(device){
      // we need again an action to get the history
      var onPropertyQuery = "devices[identifier='" + device.identifier +
"']/properties[key='http://iolite.de#on']";
      deviceAPI.action({
        request : new ActionRequest(null, null, onPropertyQuery, "getValuesOf", [
new ValueParameter(new Date().getTime()) , new ValueParameter("WEEK")]),
        success : function(historyEntries) {
          console.log("History of on-property of device:");
          historyEntries.forEach(function(entry, index) {
            console.log(index, " : ", entry);
          });
        }
      });
    });
  }
});
```

The `getValueOf` method expects two parameters, so we used two `ValueParameters` – the example reads the history going back one week since the current client's system time. Please refer to the Device API documentation (e.g. JavaDoc) for more details about the history methods and their parameter semantics.

OBSERVING DEVICES

The ModelAPI JavaScript allows you to observe changes in the API elements, similar to the `setObserver` methods in Java. Of course, since the JavaScript accesses the APIs over network, it is impossible to set an observer references directly, like in Java. Instead, a s.c. `SubscribeRequest` needs to be sent to IOLITE.

A subscription is performed for a query. The `ModelAPI` object takes care of the Websocket communication with IOLITE for you. Whenever values under the query change, the `success` callback is executed.

For the purpose of demonstration we again modify the device query example. This time, instead of a `query` or an `action`, a `subscribe` is performed:

```
deviceAPI.query({
  request : new QueryRequest(null, null, "devices"),
  success : function(devices) {
    devices.forEach(function(device){
      // subscribe to the value of the onProperty
      var onPropertyQuery = "devices[identifier='" + device.identifier +
"']/properties[key='http://iolite.de#on']/value";
      deviceAPI.subscribe({
        request : new SubscribeRequest(null, null, onPropertyQuery, null, 100),
        success : function(values) {
          console.info('Device ', device.identifier, ' is now ', values[0] ? 'on' :
'off');
        }
      });
    });
  }
});
```

The `SubscribeRequest` is initialized with the query pointing to the `value` attribute of the `on` property or each device. Whenever the `value` changes, `success` is called, where we log the new device state to console. Thus, the above example iterates over all available devices and registers for modifications of their on/off status.