IOLITE APP DEVELOPMENT GUIDE

SPECIFICATION

Last modification:	05.11.2015
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

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 (http://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:

- `org.slf4j.api`
- `org.eclipse.emf.ecore`
- `de.iolite.app.api.app-api-common`
- `de.iolite.common.lifecycle`
- `de.iolite.app.api.device-api.access`
- `de.iolite.runtime.api.runtime-api-common`
- `de.iolite.app.api.device-api`
- `de.iolite.app.api.frontend-api`
- `de.iolite.app.api.environment-api`
- `de.iolite.app.api.storage-api`
- `de.iolite.utilities.time-series`
- `de.iolite.utilities.concurrency-utils`
- `de.iolite.drivers.iolite-driver-api`
- `de.iolite.utilities.disposeable`
- `de.iolite.common.entity-identifier`
- `de.iolite.app.api.app-api-common`
- `de.iolite.common.lifecycle`
- `de.iolite.runtime.api.runtime-api-common`
- `org.slf4j.slf4j-api`

APP AS MAVEN[1] PROJECT

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

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

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):

```
<repository>
    <id>iolite-snapshots</id>
    <name>IOLITE API Snapshots</name>
    <url>http://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:

```
<!-- 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`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.5.3</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:

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

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

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"/>

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

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

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

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(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:

- `getHistorySince(long start, TimeInterval resolution, Function function)` – returns the historical values of a property from the `start` timestamp until present.
- `getHistoryBetween(long start, long end, TimeInterval resolution, Function function)` – returns a time series of historical data of a property between a `start` and `end` timestamp.
- `getHistoryOf(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.

All the mentioned `getHistory...` 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`.

For example, the code below lists all devices and then logs the history of On/Off 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<DoubleEntry> history =
powerUsage.getHistoryOf(System.currentTimeMillis(), TimeInterval.Day,
TimeInterval.Hour, Function.Average);
// iterate over the time series
for (final DoubleEntry entry : history) {
    LOGGER.info("The device used an average of {} Watt at '{}'.",
entry.getValue(), entry.getTimestamp());
}
```

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 instance are only accessed by the user who owns it.

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 subpath `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.

```
// register a handler for a non-public static resource under a specific
relative sub-path
```

```
frontendAPI.registerClasspathStaticResource("ioliteicon.png",
"de/iolite/app/example/ioliteicon.png");
// register a handler for a public static resource under a specific relative
sub-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:

```
// if you need more control of the handler (e.g. content-type), you can use
the ClasspathStaticRequestHandler
frontendAPI.registerRequestHandler("index.html", new
ClasspathStaticRequestHandler(IOLITEHTTPResponse.HTML_CONTENT_TYPE,
"de/iolite/app/example/index.html", ExampleApp.class.getClassLoader()));
});
// ... the same as public
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:

```
// finally, you can use a utility to traverse and register a set of static
resources in your JAR
FrontendAPIUtility.registerHandlers(frontendAPI,
StaticResources.scanClasspath("de/iolite/app/example/images",
getClass().getClassLoader())); //$NON-NLS-1$
// ... the same as public
FrontendAPIUtility.registerPublicHandlers(frontendAPI,
StaticResources.scanClasspath("de/iolite/app/example/images",
getClass().getClassLoader())); //$NON-NLS-1$
```

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>"); //$NON-NLS-1$
        output.append("<script
src='https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js'></scrip
t>"); //$NON-NLS-1$
        output.append("</head><body><div id='fun'><p>Example IOLITE App is
working!</p></div> <img src='logo/IOLITE-LOGO_Complex.png'
alt='IOLITE'></body>"); //$NON-NLS-1$
        try {
            // deliver the output in a response object
            return new IOLITEHTTPStaticResponse(output.toString(),
IOLITEHTTPResponse.HTML_CONTENT_TYPE);
        }
        catch (final UnsupportedEncodingException e) {
            // something went wrong, send an error response
            LOGGER.error("Failed create a response due to error: {}",
e.getMessage(), e); //$NON-NLS-1$
            return new
IOLITEHTTPStaticResponse(HTTPStatus.InternalServerError,
IOLITEHTTPResponse.GENERIC_CONTENT_TYPE);
        }
    }
});
```

The first argument for `registerRequestHandler` is the URL sub-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, nor should it rely on any pattern for its generation.
If you really need it, use `getBaseURL()`, which returns the URL of the application instance front page

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.

A special type of handler is default handler, which is always public to any client. If a client makes a request to a path, for which there is no handler, the Frontend API will call the default handler. 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>");
    output.append("</head><body><div id='fun'><p>This is an error
page</p></div></body>");
    try {
            // deliver the output in a response object
            return new IOLITEHTTPStaticResponse(output.toString(),
IOLITEHTTPResponse.HTML_CONTENT_TYPE);
        }
        catch (final UnsupportedEncodingException e) {
            // something went wrong, send an error response
            LOGGER.error("Failed create a response due to error: {}",
e.getMessage(), e); //$NON-NLS-1$
```

```
            return new
IOLITEHTTPStaticResponse(HTTPStatus.InternalServerError,
IOLITEHTTPResponse.GENERIC_CONTENT_TYPE);
        }
  }
});
```

UNREGISTERING HANDLERS

Each handler can be unregistered using its sub-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.