English

Smart API Getting Started

with examples in Java

ASEMA

Table of Contents

1. Quickstart for the impatient

Smart API is a comprehensive library for making for better API's. It is consequently not surprising that it also has quite a pile of documentation. This document is a brief intro to the main features with coding samples to try out.

Now, while we do consider this document as a whole to be short, if you'd like to skip even the brief version and would just give the first whack at it, below is a very brief and condensed set of instructions to get something going. This sample connects to a demo server, fetches some objects, and displays their details.

First the sample, after that a short explanation of what it actually does. So here we go...

1. First, create a Maven project.
2. Next, add http://maven.smart-api.io Maven repository. If you need to edit your pom.xml by hand, here's the repository spec:

```
<repository>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </releases>
    <id>SmartAPI</id>
    <url>http://maven.smart-api.io</url>
    <layout>default</layout>
</repository>
```

   and the dependency for the pom:

```
<dependency>
    <groupId>io.smart-api.smartapi</groupId>
    <artifactId>SmartAPI</artifactId>
    <version>0.0.1</version>
</dependency>
```

3. Third, save and run the script

```
//import classes from Smart API library so that you can use them in your
 code
import smartapi.common.HttpClient;
import smartapi.common.NS;
import smartapi.factory.Factory;
import smartapi.model.Entity;
import smartapi.model.PhysicalEntity;
import smartapi.model.Request;
import smartapi.model.Response;

public class QuickSample {
```

```
 String myIdentity = "http://smart-api.io/smart/examples/
Cskooterfetchersample";
 String serverUri = "http://localhost:8080/demo/smart/v1.0e1.0/access";
 //String serverUri = "http://talk.smart-api.io/demo/smart/v1.0e1.0/
access";
 HttpClient httpClient =  new HttpClient();

 /**
  * Class constructor
  */
 public QuickSample()
 {
 }

 /**
  * Creates new QuickSample instance and runs it
  * @param args
  */
 public static void main(String[] args)
 {
  QuickSample sample = new QuickSample();
  sample.fetchObject();
 }

 /**
  * Fetches an object from demo server
  */
 public void fetchObject()
 {
  httpClient.debugMode(true);
  try {
   Entity e = new Entity();
   e.addType(NS.SMARTAPI + "Skooter");
   Request req = Factory.createReadRequest(myIdentity, e);
   Response resp = httpClient.sendPost(serverUri, req);
   for ( Entity ent : resp.getFirstActivity().getEntities() ) {
    PhysicalEntity skooter = (PhysicalEntity)ent;
             System.out.println("Found a skooter " +
 skooter.getIdentifierUri());
             System.out.println("Current velocity: " +
 skooter.getVelocity().getGroundSpeed().getValue().asInt() + " " +
 NS.localName(skooter.getVelocity().getGroundSpeed().getUnit()));
             System.out.println("Current bearing: " +
 skooter.getDirection().getBearing().getValue().asInt() + " " +
 NS.localName(skooter.getDirection().getBearing().getUnit()));
             System.out.println();
   }
  } catch ( Exception e ) {
   System.err.println("Failed to fetch entity information.");
   e.printStackTrace();
  }
 }
}
```

Now, what did that sample do and what made it so special?

What the sample does is that it fetches all objects from the demo service that are of the desired type (in this case skooters) and then displays the speed and heading of each of them. Simple, but actually quite a lot happens in the background.

First, notice that while it communicated over the network, at no point was there any message content written by hand. Developers who work with simple REST APIs are often used to just writing the payload in, say, JSON as text and inserting data with string concatenation and replacement. You can do that also with Smart API, but that is tedious and you miss many of the advantages. Instead, in Smart API you create objects and let the library handle the nitty gritty of traffic.

Second, those who do use a programmatic approach, often use a library to create a document that is sent over a network. This could be a JSON document or XML document. The document is then serialized for sending and parsed at the receiver. In Smart API you work in a very similar way, except that you don't create documents, you create objects. For instance, to send a request, create a Request object, attach the details necessary to it, and serialize it.

Third, you'll notice that to get an actual value on screen, you need to dig quite deep into the structure. First you need to get the velocity, then ground speed, and then the value of that ground speed. This is intentional and a result of the simple fact that Smart API should be suitable, straight out of the box, for demanding scientific computing, geographical information systems, 3D modeling, and data automation. In such domains, it is not sufficient to just give one speed figure because velocity could be measured in relation to some other object, have components in different 3D axes, or measure not just linear velocity but also the speed of rotation. And the measurements need to have a unit, the same number in meters per second is a completely different speed than kilometers per hour. While it does take some extra effort to put such details in place, the benefit is that because this design - the so called "data modeling" - has been done by experts in advance, you get all that knoweledge of the domain as an added bonus by just following the features of the library.

2. Introduction

2.1. About this document

This document is your guide to Smart API programming. Step by step, starting from the basics you learn what Smart API is and more importantly, how you can use it in practice. After reading this document, you know how to use Smart API in your projects, and even better, you will have some working code that you use right away.

After a brief introduction, you are guided to create Smart API credentials and how to extend Smart API vocabulary with custom concepts that enable you to present exactly the information needed in your application domain. After that it is time to build your first Smart API program with your favourite programming language. Next you learn how to implement a client program that connects to a test server. Then you build your first Smart API enabled server program and a client that can fetch data from and send commands to it. Finally, you learn how to use encryption in Smart API messaging, and how to manage transactions.

2.2. What is the Smart API?

The Smart API is a technology for making better APIs for remote system and device management applications such as various IoT (Internet of Things) solutions. Technically, the Smart API is an **object centric, semantics enabled, transaction capable and secure method for transferring and storing linked data.** To understand a bit better what that is and why these features are important, let's open that up a bit:

- **Object centric**. Smart API has been designed for transferring remote objects between systems. "Objects" in this case means both the physical devices the objects represent and the programming abstractions software engineers actually use to model those devices. Objects are not only a natural way of representing physical "things" but also directly map into popular object oriented programming languages. Smart API transparently handles the details of transforming an object in memory into a datastructure transmitted over the network and back again, making the whole process easy and fast for engineers to use.

- **Semantics enabled**. Smart API builds on the principles of semantic web. The purpose of this technology is to make data exchange unambiguous and suitable for computers to deduct things from the data on behalf of users and programmers. The primary obstacle in most data exchange is that while it may work technically just fine, the data is interpreted incorrectly as different people and organizations use different terms and different measurement units for the same thing. So when for one person "length" is the dimension of a box measured in inches, for the other person the same thing is "depth" measured in centimeters. Common vocabularies i.e. semantics removes this ambiguity and makes it possible to do for instance unit conversion between inches and centimeters automatically as data is received or sent.

- **Transaction capable**. In most applications, there eventually needs to be some way to monetize the operations and data. For a long time it has been common to substitute revenue from the actual data with something else, such as advertising revenue, or apply some fixed pricing model such as a monthly fee on data services. But with the advent of Cloud Computing and Big Data, a more granular pricing model is often desired to pay just for the resources consumed. So you would pay for the amount of bytes transferred, the number of commands sent, or the number of measurements stored. Smart API transaction support is exactly for this purpose. It allows storing such details of data transfers in a way that is cryptographically protected for confidentiality and non-repudiation and serves as a ledger for invoicing.

- **Secure**. Smart API is end-to-end crypto enabled. While it is highly recommended and fully supported to use a secure links over technologies such as HTTPS, Smart API messages can further be encrypted

and signed per each message. So no matter what data communication links are in between, how the connections terminate and how the data may be stored in intermediate locations, the data remains confidential all the way to the recipient. Smart API also has built in support for authentication using OAth2 so you get the buildin blocks of security straight out of the box.

- **Linked data enabled**. Connected systems are all about - surprise - connections. Those connections link one thing to another, then to another and another and possibly back again. Such links create graphs. Understanding and being capable of processing such graphs is the core of many IoT analysis applications. In Smart API, things can be linked between objects, within messages, across messages, and in any other configuration available and necessary.

So why and when would you use Smart API? The short answer is of course when you need features listed above. The slightly longer answer is that Smart API is a highly recommended technology when you build remote control and measurement applications. True, simple get / put REST APIs over some JSON structure are the norm and in many applications sufficient - at least in the beginning. But when the application grows and it actually needs to be integrated with some larger entities, when security becomes an issue, and when you need to make sure that data is actually correct before making automation decisions, Smart API becomes an essential tool. It saves software engineers from a ton of headaches in tediously building data converters, it always stores data in an understandable format, it creates documentation for both the API and the data automatically, and it can handle data ambiguity in a professional manner. And as Smart API is free and pretty much as easy to use as alternative formats, there really is few reasons why not build a system properly from the beginning.

But there already are other API standards and tools such as Swagger/OpenAPI, why another? Well, the simple answer is: Smart API takes many of the best practices of OpenAPI such as declarative specifications and automatic generation of documents and code, but adds essential features needed by modern Big Data applications on top. Where the design of Open API originates from the API itself, Smart API's core is in the data. In a nutshell: OpenAPI is excellent is telling **how** data is transferred but lacks the functionality to tell **what** the data is. Smart API fills this gap. It supports vocabularies, links and graphs, something that OpenAPI does not, and the definition of data with proper ontologies which can be validated and tested. These are the building block for data accuracy needed in critical systems and scientific computing.

2.3. Developing applications with the Smart API

Smart API gives you a predefined structure for the data and free programming libraries. These take the burden off the application development and make creating highly sophisticated, semantic data storages a breeze. It comprises

1. An extensible datamodel that is predesigned to be able to handle data exchange and control of remote entities

2. A vocabulary of terminology that ensures all applications talk about the same things in the same way, reducing ambiguity and the risk of false processing of data

3. A programming library that makes embedding the model and the features into various IoT solutions easy

4. A supporting service infrastructure that handles issues such as autoconfiguration and security for you

The libraries are available for all major operating systems and programming languages so that you can integrate the functionality straight out of the box. To develop applications, you simply download and link the programming library - either with a package management tool or manually - and let the development tools create stub code for you to work with.

Smart API is fully extensible so that if the pre-programmed features are not sufficient for your application, you can further tweak the model and add custom details all the way down to individual triples of the RDF data that is produced.

2.4. Testing and validating a Smart API application

The development tools offer a tester against which you can test you application to make sure it runs properly and conforms to the standard. Unlike strict standards that only accept a certain fixed message structure, SEAS uses an API registry and a scoring system. You can create custom extensions as long as you register them so that other parties understand your data. A score is awarded to the design during tests and if the design deviates too much from the set model, a penalty is put on the service but it is not fully banned.

This document explains the first steps to take to get your systems running. If you are interested or required to dig deeper into the model, separate model and API documentation will guide you forward. The intended audience of this document is developers and some background in making web applications will help in following the guide.

3. Choosing the tools and the development path

Applications that use Smart API fall roughly under three categories

1. Those that connect to other applications to for instance fetch data, write values or control something like an actuator. These can be thought to be "client" applications in a classical client-server world.

2. Those that wait for connections from other applications in order to serve data or offer an API for controlling things. These are the "server" applications.

3. Those send data as broadcast to anyone interested in listening, without a specific request-response cycle. These are the "notifier" applications.

For a developer who thinks mostly in terms of network functionality, a "client" would be an HTTP or CoAP client. A "server" is then obviously an HTTP or CoAP server. A notifier is typically software that uses MQTT to publish data or alternatively WebSocket or HTTP push.

Shat said, in modern IoT systems these roles are blurred as most systems are required to sometimes act as clients, sometimes as servers. And sometimes they are just sinks or sources of notifications, making them hard to put in either category.

Whatever the overall role of the application, inside the application there still are clear blocks of code that work either as servers or clients. Smart API tools follow the same principle, offering you help in building distinct client and server - or notifier - parts. So first choose whether you want to develop the client part or the server part, the tools will then help you forward. Once done, you can come back and start from the other track to finish your application.

So once you know

- whether you are developing a client or a server

- what the main requirements of your application in this role are

- in which operating system and using which language you will be developing

you can enter the development website to get everything you need to get going.

4. Using the Smart API Developer website

4.1. Accessing the site

The Smart API Developer website can be found at http://talk.smart-api.io/developer/

Once at the site, choose the correct development desktop for your needs

- Service Desktop, if you are coding server-side functionality
- Client Desktop, if you are coding client-side functionality
- Notifier Desktop, if you are coding notifier functionality

4.2. Creating a username and password

You can use the Smart API Developer site either anonymously or by signing in. Access in both ways is absolutely free.

Anonymous access gives you the basics of getting acquainted with the philosophy of Smart API and initial application development. If you are not signed in, you can browse the ontologies (i.e. data specifications) available for the API and generate sample code based on that. That is sufficient to get started.

Being logged in gives you additional features that help in developing larger applications and managing devices and security.

- First, you can save whatever you do on the site so you don't have to start all over again when you return to work.
- Second, you can generate test tokens that can be used to run the automatic protocol testers agains the Smart API standard test end point - this greatly helps in testing that your application actually works as planned.
- Third, you can manage the public encryption keys for your services and devices, making it possible for other parties to send your services data in encrypted format.
- Fourth, you can manage your own vocabularies and data definitions. This is essential if you create applications for integration with other systems.

To sign up to the developer tools, simply click on "Sign up" at the right top corner and then fill in your details. A thing to pay attention to is the so called prefix, which is organization specific. If your organization already has a prefix assigned you you'd like to reserve a specific one, fill that into the corresponding field. Note that the prefix needs to be globally unique (you cannot have the same prefix as someone else). If you do not want to pick a prefix, leave the field blank and fill in the company name. A prefix will be generated automatically for you.

4.3. Editing vocabularies

4.3.1. What is a vocabulary

A vocabulary is a set of definitions of variables you use for your data. The problem it solves is a common one: if you have some variable you want to put into your data, how would you name it so that everyone else understands it? Say for example, you want to store the speed at which a vehicle drives. Would you call this "speed" or "velocity" or "groundSpeed" or what is it?

A widely used solution for the problem is to provide an API doc. So you have some separate document that says there is a property called "velocity" which measures the speed of a vehicle in relation to the road. But this is quite tedious. First, you need to distribute that API doc. And second, there is no way to automate the mapping of APIs so everyone who uses it needs to go through the process of manually doing everything. And once you're past that hurdle, comes the question of units. Is that speed expressed in mph, kph, m/s or what is it?

With a vocabulary, you jointly agree on such terms. So when someone measures speed, let's all call it "groundSpeed" and measure it in meters per second. While an improvement, there are still problems with this approach. First, what if there are two parties that do call that property groundSpeed but use it with different meanings and different units?

To solve the issues, a proper vocabulary first separates different definitions by domain. Technically this means that the term is expressed with a URL, say http://www.commonterms.org/velocity#groundSpeed. That full URL reduces the risk of conflicting definitions. The URL is usually replaced by a prefix. For example `commonTerms:groundSpeed`. And if you have a differing meaning for it, that would be `myTerms:groundSpeed`. The useful thing about URL's is that you can actually point with them somewhere, in the case of vocabularies what it points to is the definition of the term. So if someone browses http://www.commonterms.org/velocity#groundSpeed, that link would return text that explains what groundSpeed is. No need for separate API docs, everything is conveniently online.

Further, vocabularies can do inheritance and classification. This is very similar to inheritance in object oriented programming. Let's for instance assume you are making objects that are all boxes. So the basic type would be for example `myproducts:box`. Some of these boxes are designed to be useless toys, some useful tools. Which means the box has two subcategories `myproducts:uselessbox` and `myproducts:useful`. Let's further divide the useless boxes to `myproducts:toybox` and `myproducts:decorativebox` and the useful boxes have a subcategory `myproducts:toolbox`. In plan data exchange such classifications have limited meaning but when it comes to search, they are invaluable. For instance if you want to find all useless boxes, you don't need to search for both `myproducts:toybox` and `myproducts:decorativebox` but instead can perform one search with `myproducts:uselessbox`. Automatically processing such hierarchies is the core of semantic tools, they automatically understand the classifications and can greatly help in data management.

4.3.2. Extending your own vocabulary

Once you know that you need a vocabulary for something that you'd like to express as data, you add it using the tools on the Smart API developer site. Do note that the existing vocabularies already cover a huge amount of things, especially in the domain of measurements and units. So it is always a good idea to check whether something exists before adding one of your own.

To add some new concept to a vocabulary, click on the vocabulary management tool icon on the left toolbar and enter the new concept. Concepts exist in many categories, for instance quantities that define what you are measuring (e.g. LiquidVolume), units that say in which unit that measurement is (e.g. Liter) and types of things (e.g. monocycle, bicycle, tricycle).

Once happy with your entry, click on submit to add it. The entry will be automatically added to the vocabulary directory and will appear in the search functionality on the developer site as well as the Smart API IDE plugins you cna use to easily write compatible code with your favorite IDE.

4.4. Creating code for a service

To provide a new service, follow the steps below.

1. At the desktop, describe the inputs and the outputs of the service by selecting the individual items to the workspace.

2. Save your service description.

3. Generate program code for your software by clicking the autogenerate icon.
4. Copy the generated code to your own application.
5. Obtain and link the SeasObjects library to your application.
6. Build and run your application.

4.5. Creating code for a client

To call an existing service, follow the steps below.

1. At the client desktop, first find a service using the provided search tool. The tool will analyze the outputs the service offers and give you a list of the available options.
2. Select the output items you are interested in and generate program code for your software by clicking the autogenerate icon.
3. Copy the code to your own application.
4. Obtain and link the SeasObjects library to your application, if not done so yet.
5. Run and test your application.

5. Your first Smart API program

It is time to get you powered up and start programming. To write your first program, you need to download the library, import it and write simple code for the 'HelloSmart' program.

5.1. Download Smart API library

Using Smart API is made easy with libraries that abstract away all the semantic details you do not need to worry about. Library is currently available for C++, C#, Java and Python.

Download library from: http://talk.smart-api.io/develop/#download

5.2. Choosing IDE

To make writing code more pleasant, you can install an Integrated Development Environment (IDE) that will help you with many tasks, such as importing libraries into your projects. Smart API libraries were developed using Eclipse IDE, but you can choose any other, or use a simple text editor if you wish.

Download Eclipse IDE from: https://eclipse.org/

5.3. Importing libraries

Smart API has several dependencies to other libraries that need to be available in classpath in case features that utilize those libraries are used by Smart API. For the examples in this document, add these libraries into classpath:

- `smartapi.jar`
- `catalina.jar`
- `jsonorg.jar`
- `gson-2.4.jar`
- `collection-0.6.jar`
- `commons-cli-1.3.jar`
- `commons-codec-1.9.jar`
- `commons-csv-1.0.jar`
- `commons-io-2.4.jar`
- `commons-lang3-3.3.2.jar`
- `httpclient-4.2.6.jar`
- `httpclient-cache-4.2.6.jar`
- `httpcore-4.4.5.jar`
- `jackson-annotations-2.3.0.jar`
- `jackson-core-2.3.3.jar`
- `jackson-databind-2.3.3.jar`

- jcl-over-slf4j-1.7.20.jar
- jena-arq-3.1.0.jar
- jena-base-3.1.0.jar
- jena-cmds-3.1.0.jar
- jena-core-3.1.0.jar
- jena-iri-3.1.0.jar
- jena-shaded-guava-3.1.0.jar
- jena-tdb-3.1.0.jar
- jsonld-java-0.7.0.jar
- libthrift-0.9.2.jar
- log4j-1.2.17.jar
- slf4j-api-1.7.20.jar
- slf4j-log4j12-1.7.20.jar
- xercesImpl-2.11.0.jar
- xml-apis-1.4.01.jar

When writing the program, you only need to import classes that you are directly using. For this example, following imports need to be made in your code:

```
// import classes from Smart API library so that you can use them in your
 code
import smartapi.common.RESOURCE;
import smartapi.model.Device;
import smartapi.model.ValueObject;
```

5.4. Writing code

Your first program is called HelloSmart. Your code should look like this:

```
// import classes from Smart API library so that you can use them in your
 code
import smartapi.common.RESOURCE;
import smartapi.model.Device;
import smartapi.model.ValueObject;

// name your class HelloSmart and implement a method smartAPITest.
public class HelloSmart {

 /**
  * Class constructor
  */
```

```
 public HelloSmart()
 {
 }

 /**
  * Runs HelloSmart test.
  * Creates one Device object that has one ValueObject
  * and prints the Device object out in Turtle RDF format.
  */
 public void smartAPITest()
 {
  // identity is a URI that is unique and only used by one particular object
 (this device)
  String deviceIdentity = "http://smart-api.io/smart/examples/
CHelloSmartDevice";
  // create new Device with given identity
  Device radio = new Device(deviceIdentity);
  // create new ValueObject that represents weight of 3.6 kg
  ValueObject weight = new ValueObject(RESOURCE.WEIGHT, RESOURCE.KILOGRAM,
 3.6);
  // set weight ValueObject as radio Device weight
  radio.setWeight(weight);
  // print out radio in Turtle RDF format to console
  radio.turtlePrint();
 }
}
```

Finally create a main method so you can run the code directly with this class. Complete code of your program should look like this:

```
// import classes from Smart API library so that you can use them in your
 code
import smartapi.common.RESOURCE;
import smartapi.model.Device;
import smartapi.model.ValueObject;

public class HelloSmart {

 /**
  * Class constructor
  */
 public HelloSmart()
 {
 }

 /**
  * Creates new HelloSmart instance and runs it
  * @param args
  */
 public static void main(String[] args)
 {
  HelloSmart helloSmart = new HelloSmart();
  helloSmart.smartAPITest();
 }
```

```
 /**
  * Runs HelloSmart test.
  * Creates one Device object that has one ValueObject
  * and prints the Device object out in Turtle RDF format.
  */
 public void smartAPITest()
 {
  // identity is a URI that is unique and only used by one particular object
 (this device)
  String deviceIdentity = "http://smart-api.io/smart/examples/
CHelloSmartDevice";
  // create new Device with given identity
  Device radio = new Device(deviceIdentity);
  // create new ValueObject that represents weight of 3.6 kg
  ValueObject weight = new ValueObject(RESOURCE.WEIGHT, RESOURCE.KILOGRAM,
 3.6);
  // set weight ValueObject as radio Device weight
  radio.setWeight(weight);
  // print out radio in Turtle RDF format to console
  radio.turtlePrint();
 }
}
```

Running the program will output the device data to console. Do not worry if you do not understand what the printed RDF means. It is here just for testing purposes. Output should look like this:

```
@prefix smartapi: <http://smart-api.io/ontology/1.0/smartapi#> .
@prefix qudt:  <http://data.nasa.gov/qudt/owl/qudt#> .
@prefix unit:  <http://data.nasa.gov/qudt/owl/unit#> .
@prefix quantity: <http://data.nasa.gov/qudt/owl/quantity#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

<http://smart-api.io/smart/examples/CHelloSmartDevice>
        a                 smartapi:Device ;
        smartapi:weight   [ a                   smartapi:ValueObject ;
                            rdf:value           "3.6"^^xsd:double ;
                            qudt:quantityKind   quantity:Weight ;
                            qudt:unit           unit:Kilogram
                          ] .
```

Congratulations, you have just created your first Smart API program!

6. Creating Smart API client program

This section will demonstrate how to create a simple Smart API client program. To make testing your client easier, a test server has been set up to process your requests.

The program is very simple. It will first fetch entity information from the server and print out all the ValueObjects that the entity holds. Then it will command a new value for one of the ValueObjects and again print out the ValueObjects that the server returns. Finally, it will send again the same request as in the first time, to see if the command has really taken effect on the server.

6.1. Server information

Test server has not registered its description in Smart API registry and we thus cannot find there required information for making requests. For this example, we assume that we already have the following information available.

- Server URI: `https://demo.iot.asema.com:443/smart/v1.0e1.0/access`
- Identifier of the entity on server: `http://demo.iot.asema.com/objects/C7f9c5aea0bf31f97483a8ab3e6579b390c982291`

6.2. Fetching entity information

Using the service URI and the known identity of the entity on the server, you can easily fetch the whole entity information. Creating a right kind of request is simply handled by the Factory.createReadRequest() method. Your code should look like this:

```
/**
 * Fetch one entity from test Smart API server
 */
public void fetchEntity()
{
 // your identity uri
 String myIdentity = "http://smart-api.io/smart/examples/CSmartAPIClient";
 // interface uri of the API that the server provides
 String serverUri = "https://demo.iot.asema.com:443/smart/v1.0e1.0/access";
 // identity of the entity that you are requesting
 String serverEntityIdentity = "http://demo.iot.asema.com/objects/
C7f9c5aea0bf31f97483a8ab3e6579b390c982291";
 // HTTP client
 HttpClient client = new HttpClient();
 try {
  // create standard read request object with your identity as sender
  Request req = Factory.createReadRequest(myIdentity, new
 Entity(serverEntityIdentity));
  // make request
  Response resp = client.sendPost(serverUri, req);
  // print out  the values of the ValueObject that the response holds for the
 requested entity
  for ( ValueObject vo :
 resp.getActivities().get(0).getEntities().get(0).getValueObjects() ) {
   System.out.println("*** ValueObject ***");
   System.out.println("Quantity: " + (vo.hasQuantity() ?
 NS.localName(vo.getQuantity()) : " not defined"));
   System.out.println("Unit: " + (vo.hasUnit() ? NS.localName(vo.getUnit()) :
 " not defined"));
```

```
    System.out.println("Value: " + (vo.hasValue() ? vo.getValueAsString() : "
 not defined"));
    System.out.println();
   }
 } catch ( Exception e ) {
   System.out.println("Failed to fetch entity information.");
   e.printStackTrace();
 }
}
```

6.3. Sending a command

Code for sending a write request is very similar to reading. You only need to change the value in the ValueObject you received in the previous method, and use Factory.createWriteRequest() for building the request object. Your code should look like this:

```
/**
 * Request test server to change temperature to a different value
 */
public void commandTemperature(ValueObject temperature)
{
 // your identity uri
 String myIdentity = "http://smart-api.io/smart/examples/CSmartAPIClient";
 // interface uri of the API that the server provides
 String serverUri = "https://demo.iot.asema.com:443/smart/v1.0e1.0/access";
 // identity of the entity that you are requesting
 String serverEntityIdentity = "http://demo.iot.asema.com/objects/
C7f9c5aea0bf31f97483a8ab3e6579b390c982291";
 // HTTP client
 HttpClient client = new HttpClient();
 try {
  // set new temperature value
  Random rnd = new Random();
  int newValue;
  // generate random value that differs from the old value
  for ( newValue = -1; newValue < 0 || newValue ==
 temperature.getValueAsInt(); newValue = rnd.nextInt(3000) ) {};
  temperature.setValue(newValue);
  // create standard write request object with your identity as sender
  Request req = Factory.createWriteRequest(myIdentity, new
 Entity(serverEntityIdentity), temperature);
  // make request
  Response resp = client.sendPost(serverUri, req);
  // print out  the values of the ValueObject that the response holds for the
 requested entity
  for ( ValueObject vo :
 resp.getActivities().get(0).getEntities().get(0).getValueObjects() ) {
   System.out.println("*** ValueObject ***");
   System.out.println("Quantity: " + (vo.hasQuantity() ?
 NS.localName(vo.getQuantity()) : " not defined"));
   System.out.println("Unit: " + (vo.hasUnit() ? NS.localName(vo.getUnit()) :
 " not defined"));
   System.out.println("Value: " + (vo.hasValue() ? vo.getValueAsString() : "
 not defined"));
```

```
    System.out.println();
   }
  } catch ( Exception e ) {
   System.out.println("Failed to command new value for entity valueobject.");
   e.printStackTrace();
  }
 }
```

6.4. Running the program

Finally create a main method so you can run the code directly with this class. Complete code of your program should look like this:

```
package smartapiclient;

import java.util.Random;

import smartapi.common.HttpClient;
import smartapi.common.NS;
import smartapi.common.RESOURCE;
import smartapi.factory.Factory;
import smartapi.model.Entity;
import smartapi.model.Request;
import smartapi.model.Response;
import smartapi.model.ValueObject;

public class SmartAPIClient {

 // your identity uri
 String myIdentity = "http://smart-api.io/smart/examples/CSmartAPIClient";
 // interface uri of the API that the server provides
 String serverUri = "https://demo.iot.asema.com:443/smart/v1.0e1.0/access";
 // identity of the entity that you are requesting
 String serverEntityIdentity = "http://demo.iot.asema.com/objects/
C7f9c5aea0bf31f97483a8ab3e6579b390c982291";
 // HTTP client
 HttpClient client = new HttpClient();
 // temperature valueobject
 ValueObject temperature = null;

 /**
  * Class constructor
  */
 public SmartAPIClient()
 {
 }

 /**
  * Creates new SmartAPIClient instance and runs it
  * @param args
  */
 public static void main(String[] args)
 {
  SmartAPIClient smartAPIClient = new SmartAPIClient();
```

```
 // read request
 smartAPIClient.fetchEntity();
 // write request
 smartAPIClient.commandTemperature();
 // read again to make sure that writing took effect
 smartAPIClient.fetchEntity();
}

/**
 * Fetch one entity from test Smart API server
 */
public void fetchEntity()
{
 System.out.println("Running fetch entity:");
 try {
  // create standard read request object with your identity as sender
  Request req = Factory.createReadRequest(myIdentity, new
Entity(serverEntityIdentity));
  // make request
  Response resp = client.sendPost(serverUri, req);
  // print out  the values of the ValueObject that the response holds for
the requested entity
  for ( ValueObject vo :
resp.getActivities().get(0).getEntities().get(0).getValueObjects() ) {
   System.out.println("*** ValueObject ***");
   System.out.println("Quantity: " + (vo.hasQuantity() ?
NS.localName(vo.getQuantity()) : " not defined"));
   System.out.println("Unit: " + (vo.hasUnit() ?
NS.localName(vo.getUnit()) : " not defined"));
   System.out.println("Value: " + (vo.hasValue() ? vo.getValueAsString() : "
not defined"));
   System.out.println();

   // save standard temperature valueObject to be used in the command
   if ( vo.hasQuantity() && vo.getQuantity().equals(RESOURCE.TEMPERATURE) )
{
    this.temperature = vo;
   }
  }
 } catch ( Exception e ) {
  System.out.println("Failed to fetch entity information.");
  e.printStackTrace();
 }
}

/**
 * Request test server to change temperature to a different value
 */
public void commandTemperature()
{
 System.out.println("Running command temperature:");
 try {
  // set new temperature value
  Random rnd = new Random();
  int newValue;
  // generate random value that differs from the old value
  for ( newValue = -1; newValue < 0 || newValue ==
this.temperature.getValueAsInt(); newValue = rnd.nextInt(3000) ) {};
```

```
    this.temperature.setValue(newValue);
    // create standard write request object with your identity as sender
    Request req = Factory.createWriteRequest(myIdentity, new
 Entity(serverEntityIdentity), this.temperature);
    // make request
    Response resp = client.sendPost(serverUri, req);
    // print out  the values of the ValueObject that the response holds for
 the requested entity
    for ( ValueObject vo :
 resp.getActivities().get(0).getEntities().get(0).getValueObjects() ) {
     System.out.println("*** ValueObject ***");
     System.out.println("Quantity: " + (vo.hasQuantity() ?
 NS.localName(vo.getQuantity()) : " not defined"));
     System.out.println("Unit: " + (vo.hasUnit() ?
 NS.localName(vo.getUnit()) : " not defined"));
     System.out.println("Value: " + (vo.hasValue() ? vo.getValueAsString() : "
 not defined"));
     System.out.println();
    }
   } catch ( Exception e ) {
    System.out.println("Failed to command new value for entity valueobject.");
    e.printStackTrace();
   }
  }
}
```

Running the program will output the ValueObject data to console. From the output you can if the command really changed the value on the server. Output should look like this:

```
Running fetch entity:
*** ValueObject ***
Quantity: http://www.smart-api.io/schema/quant/temperature
Unit: http://www.nasa.gov/ontology/unit/fahrenheit
Value:  not defined

*** ValueObject ***
Quantity: Temperature
Unit: DegreeCelsius
Value: 293

Running command temperature:
*** ValueObject ***
Quantity: Temperature
Unit: DegreeCelsius
Value: 2551

Running fetch entity:
*** ValueObject ***
Quantity: http://www.smart-api.io/schema/quant/temperature
Unit: http://www.nasa.gov/ontology/unit/fahrenheit
Value:  not defined

*** ValueObject ***
Quantity: Temperature
Unit: DegreeCelsius
```

```
Value: 2551
```

7. Creating Smart API server program

This section will demonstrate how to create a simple Smart API server program. To make testing your server easier, we will also create a test program that connects to your server for reading and writing values.

7.1. What does a Smart API server do?

Once receiving request message, a typical Smart API server will parse the message and build a Request Object out of it. It then composes response message according to the following steps:

- 1. use factory to create a Response object
- 2. add Activity objects to Response
- 3. add Entity objects to the Activity
- 4. add necessary properties to Entity objects, based on application.
- 5. serialize this Response object to string (using Tools.serializeResponse function), and send out through wire.

The aforementioned steps can be illustrated in the following sample code.

```
protected void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
 PrintWriter out = response.getWriter();
 String reqBody;
 String contentTypeHeader = request.getContentType();
 String contentType = HttpMessage.getMainRdfContentType(contentTypeHeader);
 String serialization = SERIALIZATION.fromContentType(contentType);
 if ( serialization.equals(SERIALIZATION.UNKNOWN) ) {
  serialization = SERIALIZATION.DEFAULT;
  contentType = SERIALIZATION.toContentType(serialization);
 }
 response.setContentType(contentType);
 try {
  reqBody =
 request.getReader().lines().collect(Collectors.joining(System.lineSeparator()));
 } catch (Exception e) {
  out.println(Tools.toString(Factory.createParseErrorResponse(myIdentity,
 "Unable to read request body"),
    serialization));
  return;
 }
 Request req;
 try {
  req = Tools.parseRequest(reqBody, contentType);
  Response resp = Factory.createResponse(req, myIdentity);
  Activity a = new Activity();
  PhysicalEntity skooter1 = new
 PhysicalEntity(Tools.createIdentifierUri(domain, serviceIdentifier,
 skooter1Identifier));
  PhysicalEntity skooter2 = new
 PhysicalEntity(Tools.createIdentifierUri(domain, serviceIdentifier,
 skooter2Identifier));
  skooter1.addType(NS.SMARTAPI + "Skooter");
  skooter2.addType(NS.SMARTAPI + "Skooter");
```

```
  Velocity v1 = new Velocity();
  Velocity v2 = new Velocity();
  Direction d1 = new Direction();
  Direction d2 = new Direction();
  v1.setGroundSpeed(new ValueObject(RESOURCE.SPEED,
 RESOURCE.KILOMETERPERHOUR, 54));
  v2.setGroundSpeed(new ValueObject(RESOURCE.SPEED,
 RESOURCE.KILOMETERPERHOUR, 41));
  d1.setBearing(new ValueObject(null, RESOURCE.DEGREEANGLE, 183));
  d2.setBearing(new ValueObject(null, RESOURCE.DEGREEANGLE, 164));
  skooter1.setVelocity(v1);
  skooter2.setVelocity(v2);
  skooter1.setDirection(d1);
  skooter2.setDirection(d2);
  a.addEntity(skooter1);
  a.addEntity(skooter2);
  resp.addActivity(a);
  out.println(Tools.serializeResponse(resp, serialization).getLeft());
 } catch (Exception e) {
  e.printStackTrace();
  out.println(Tools.toString(
    Factory.createServerErrorResponse(myIdentity, "Error while handling
 request. " + e.getMessage()),
    serialization));
  return;
 }
}
```

7.2. Running the program

To complete the program, you can implement it, for instance, as a servlet. Complete servlet code should look like this:

```
package serverforimpatient;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.stream.Collectors;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import smartapi.common.HttpMessage;
import smartapi.common.NS;
import smartapi.common.RESOURCE;
import smartapi.common.SERIALIZATION;
import smartapi.common.Tools;
import smartapi.factory.Factory;
import smartapi.model.Activity;
import smartapi.model.Direction;
import smartapi.model.PhysicalEntity;
```

```
import smartapi.model.Request;
import smartapi.model.Response;
import smartapi.model.ValueObject;
import smartapi.model.Velocity;

/**
 * Servlet implementation class ServerForImpatient
 */
@WebServlet("/smart/v1.0e1.0/access")
public class ServerForImpatient extends HttpServlet {
 private static final long serialVersionUID = 1L;
 private final String domain = "smart-api.io";
 private final String serviceIdentifier = "CexampleServerForImpatient";
 private final String skooter1Identifier = "skooter1";
 private final String skooter2Identifier = "skooter2";
 // identifier uri for this service
 private String myIdentity= Tools.createIdentifierUri(domain,
 serviceIdentifier);

 /**
  * @see HttpServlet#HttpServlet()
  */
 public ServerForImpatient()
 {
  super();
 }

 // called on servlet initialization on tomcat start
 public void init() throws ServletException
 {
  System.out.println("** ServerForImpatient example started ** ");
 }

 /**
  * Handle requests to get data from the server
  * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
 response)
  */
 protected void doPost(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  String reqBody;
  String contentTypeHeader = request.getContentType();
  String contentType = HttpMessage.getMainRdfContentType(contentTypeHeader);
  String serialization = SERIALIZATION.fromContentType(contentType);
  if ( serialization.equals(SERIALIZATION.UNKNOWN) ) {
   serialization = SERIALIZATION.DEFAULT;
   contentType = SERIALIZATION.toContentType(serialization);
  }
  response.setContentType(contentType);
  try {
   reqBody =
 request.getReader().lines().collect(Collectors.joining(System.lineSeparator()));
  } catch (Exception e) {
   out.println(Tools.toString(Factory.createParseErrorResponse(myIdentity,
 "Unable to read request body"),
     serialization));
   return;
```

```
   }
   Request req;
   try {
    req = Tools.parseRequest(reqBody, contentType);
    Response resp = Factory.createResponse(req, myIdentity);
    Activity a = new Activity();
    PhysicalEntity skooter1 = new
  PhysicalEntity(Tools.createIdentifierUri(domain, serviceIdentifier,
  skooter1Identifier));
    PhysicalEntity skooter2 = new
  PhysicalEntity(Tools.createIdentifierUri(domain, serviceIdentifier,
  skooter2Identifier));
    skooter1.addType(NS.SMARTAPI + "Skooter");
    skooter2.addType(NS.SMARTAPI + "Skooter");
    Velocity v1 = new Velocity();
    Velocity v2 = new Velocity();
    Direction d1 = new Direction();
    Direction d2 = new Direction();
    v1.setGroundSpeed(new ValueObject(RESOURCE.SPEED,
  RESOURCE.KILOMETERPERHOUR, 54));
    v2.setGroundSpeed(new ValueObject(RESOURCE.SPEED,
  RESOURCE.KILOMETERPERHOUR, 41));
    d1.setBearing(new ValueObject(null, RESOURCE.DEGREEANGLE, 183));
    d2.setBearing(new ValueObject(null, RESOURCE.DEGREEANGLE, 164));
    skooter1.setVelocity(v1);
    skooter2.setVelocity(v2);
    skooter1.setDirection(d1);
    skooter2.setDirection(d2);
    a.addEntity(skooter1);
    a.addEntity(skooter2);
    resp.addActivity(a);
    out.println(Tools.serializeResponse(resp, serialization).getLeft());
   } catch (Exception e) {
    e.printStackTrace();
    out.println(Tools.toString(
      Factory.createServerErrorResponse(myIdentity, "Error while handling
  request. " + e.getMessage()),
      serialization));
    return;
   }
  }
}
```

You can test this tiny Smart API server with the client in Chapter 1 "Quickstart for the impatient". Do not forget to replace serverUri with this server's own URL "http://localhost:8080/ServerForImpatient/smart/v1.0e1.0/access".

8. Obtaining the Smart API SDK library

The code you get from the developer website will use the Smart API library which takes care of the majority of details in handling the data. You can, in principle, also use the Smart API data model by writing the model directly, but the library will significantly reduce development effort and speed up the process.

The library is available in all major programming languages. Direct links for downloading library, including source packages, are available at the developer website.

In addition to the manual downloads, Smart API is available in common package repositories.

- Java - Maven. To install, add the SEAS Maven repository and the SeasObjects dependency into your project (pom) as show below.
- .net/C# - NuGet. To install, go to NuGet package management in Visual Studio, search for SeasObjects and click on install.
- Python - pip. To install, use the Python package index by issuing at command line `pip install SeasObjects_for_Python`.

For Maven, add these two XML snippets into your pom.xml file

To link the repository

```
<repository>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </releases>
    <id>SeasObjects</id>
    <url>http://maven.seas.asema.com</url>
    <layout>default</layout>
</repository>
```

To add the dependency

```
<dependency>
    <groupId>com.asema.seas</groupId>
    <artifactId>SeasObjects</artifactId>
    <version>0.0.1</version>
</dependency>
```

9. Testing your software

Once you have integrated SEAS code to your software for either providing or consuming a service, you can run it against the SEAS Tester website.

To test your newly created service, follow the steps below.

1. Go to the SEAS Tester website. http://seas.asema.com/tester/ -> Test my service
2. Enter the URI of your published service.
3. Run the test.
4. Analyze the results and, if needed, improve the service as adviced.

If you are using an existing service. Follow the steps below to test your newly created client code.

1. Go to the SEAS Tester website. http://seas.asema.com/tester/ -> Test my client
2. Request a test token from the tester.
3. Include the test token into your client code as X-Seas-Test-Token header.
4. Change the server address in your code to SEAS Tester URI http://seas.asema.com/tester/
5. Run the client code.
6. Analyze the results on the website and, if needed, improve the client code s advised.

10. Registrating services and other entities

Once the software has been developed and passes the tests, it would of course be nice to announce it to others and make it possible for other parties to find it and configure their part accordingly to use it. The purpose of the SEAS registry is to help in this.

So once you have integrated SEAS service code to your software and tested it against the SEAS Tester website, you need to generate its registration code. In practice the registration takes place thorugh a software component called a Registration Agent. What you need to do is to describe what your application does to the agent, and the agent will then do the actual communication with the registry.

After registration the SEAS registry will contain a "recipe" of your software API which allows others to create their parts of the communication with the API. You may also register other details of your solution, such as the location and the price for usage. Especially if your application runs a physical device, the geographical search capabilities of the registry will allow others to find suitable devices to use and share in the vicinity of their own operations.

To registrate your service, follow the steps below.

1. Go to the SEAS Developer website. http://seas.asema.com/developer/ -> ServiceDesktop
2. Load your service description (saved earlier when generating the service code).
3. Generate program code for your software.
4. Copy the code to your own application.
5. Run your application.

11. Authoring and modifying code

The SEAS Developer tools will create you sample stub code for many purposes and should help in the majority of basic API needs. You are of course free to modify and adapt the code to fit your particular application. For this purpose, this chapter explains the anatomy of four basic operations and their sample code

1. Creating a service API
2. Registering a service API for the search directory
3. Finding a service API from the search directory.
4. Connecting to a service API in the search directory.

11.1. Code for creating a service API

This example creates an API that responds to POST calls and provides data - in this case weather forecasts - as a response. The responder has been programmed with Java and establishes a standard servlet responder that can be deployed as such to e.g. Tomcat.

For examples in other languages, please visit the documentation section of the developer portal.

Let's first set up a standard HTTP POST responder. Does not have to be a servlet, anything that gets you the headers and the payload from an HTTP POST would do.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
 ... the rest of the example code should be placed here ...
}
```

Next Extract the headers we are interested in, especially the method and callId

```
 StringBuffer url = request.getRequestURL();
 String responseType = request.getHeader("Accept");
 String method = request.getHeader("x-seas-method");
 String callId = request.getHeader("x-seas-callId");
 String contentTypeHeader = request.getContentType();

 String serialization;
 PrintWriter out = response.getWriter();

 // Be polite to the caller, respond back with the same callId
 response.setHeader("x-seas-callId", callId);

 // If response type not explicitly defined, respond back with the same
 // content type as in the request.
 if ( responseType == null || responseType.equals("") ) {
  responseType = HttpMessage.getMainRdfContentType(contentTypeHeader);
 }

 // Set response serialization. This is important as RDF can
 // be encoded into many different formats (JSON-LD, Turtle, RDF-XML)
```

```
serialization = SERIALIZATION.fromContentType(responseType);
if ( serialization.equals(SERIALIZATION.UNKNOWN) ) {
 serialization = SERIALIZATION.DEFAULT;
}
response.setContentType(SERIALIZATION.toContentType(serialization));
if ( method == null ) {
 System.out.println("Missing x-seas-method header.");

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
  "Missing x-seas-method header."), serialization));
  return;
}
```

The standard defines that for compliance, all APIs should respond from the same URL path. Note that a non-standard path can also be registered

```
if (url.toString().endsWith("seas/v1.0e1.0/access/")) {
 if (method.equalsIgnoreCase("Request")) {
```

Fetch the payload from the HTTP request and parse it. The Tools module of SeasObjects can do this automatically. So create one Request object and let the Tools give it content.

```
 Request req;
 try {
  String body =
request.getReader().lines().collect(Collectors.joining(System.lineSeparator()));
  req = Tools.parseRequest(body, contentTypeHeader);
 } catch ( Exception e ) {
  System.out.println("Unable to interpret message body as SEAS RDF
Request.");
  out.println(Tools.toString(Factory.createParseErrorResponse(myIdentity,
"Unable to interpret message body as SEAS RDF Request."), serialization));

  return;
 }
```

Now we have the RDF data as objects. Start processing it. Each Request should have inputs that the caller has given. Read them in.

```
 ArrayList<Input> inputs = null;
 try {
  inputs = req.getActivities().get(0).getInputs();
 } catch ( Exception e ) {
  System.out.println("Unable to find inputs from the request.");

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
    "Could not find input parameters. Please, check the request
structure."),
    serialization));
```

```
   return;
  }

  if ( inputs.size() != 1 ) {
   System.out.println("Wrong number of inputs. Expected 1, found " +
inputs.size() + ".");

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
     "Wrong number of inputs. Expected 1, found " + inputs.size() + ".
Please, check the request structure."),
     serialization));
   return;
  }
  Input input = inputs.get(0);

  // OK, got the parameters. Our weather service is defined to offer
forecasts for a certain
  // location (given as coordinates). It takes start and end times as
parameters and returns
  // the output values the client has requested
  Coordinates coordinates = null;
  Date start = null;
  Date end = null;
  boolean solarAccess = false;
  boolean cloudiness = false;
  boolean temperature = false;
  String temperatureUnit = null;
  int numberOfOutputQuantities = 0;

  // read start and end times
  if ( input.hasTemporalContext() ) {
   TemporalContext tcx = input.getTemporalContext();
   if ( tcx.hasStart() ) {
    start = tcx.getStart().asDate();
   }
   if ( tcx.hasEnd() ) {
    end = tcx.getEnd().asDate();
   }
  }

  // read coordinates
  coordinates = (Coordinates)input.getFirst("coordinates").asObj();
  if ( coordinates == null ) {
   System.out.println("Missing coordinates parameter.");

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
     "Could not find input parameters. Please, check the request
structure."),
     serialization));
   return;
  }

  // read output values
  if ( input.hasOutputValue() ) {
   for ( ValueObject vo : input.getOutputValues() ) {
    String quantity = vo.getQuantity();
    String unit = vo.getUnit();
    switch (quantity) {
```

```
    case RESOURCE.SOLARACCESS:
     solarAccess = true;
     numberOfOutputQuantities++;
     break;
    case RESOURCE.CLOUDINESS:
     cloudiness = true;
     numberOfOutputQuantities++;
     break;
    case RESOURCE.TEMPERATURE:
     temperature = true;
     temperatureUnit = unit;
     numberOfOutputQuantities++;
     break;
    default:
     System.out.println("Unsupported quantity requested: " + quantity);

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
      "Unsupported quantity requested: " + quantity),
      serialization));
     return;
    }
   }
  }

  // if unit for temperature is not given, use celsius
  if ( temperatureUnit == null ) {
   temperatureUnit = RESOURCE.DEGREECELSIUS;
  }

  if( numberOfOutputQuantities < 1 ) {
   System.out.println("Output values are not specified.");

out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
     "Please, specify requested output quantities in the request."),
     serialization));
   return;
  }
```

At this point we have everything we need from the request. Now we can create a response. The Factory will handle that neatly.

```
  Response resp = null;
  try {
   resp = Factory.createResponse(myIdentity);
  } catch ( Exception e ) {
   System.out.println("Unable to generate response object.");
   out.println(Tools.toString(Factory.createServerErrorResponse(myIdentity,
     "Unable to generate response object."),
     serialization));
   return;
  }
```

In SEAS, all entities perform "activities" such as measurements, controls, payments, etc When a response is given, it is bound to one such activity. So first create an activity, the rest of the data goes inside it as outputs. For simplicity, this Activity in our example has no further identification, it is just a wrapper for data.

```
Activity activity = new Activity();
// This activity produces an output so make one.
Output output = new Output();
// add categories for the output data
output.addCategory(RESOURCE.WEATHER);
output.addCategory(RESOURCE.FORECAST);

// default start is today
if ( start == null ) {
 start = new Date();
}
// default end is start + one week
if ( end == null || end.before(start) ) {
 end = (Date)start.clone();
 Factory.createDuration(0, 0, 7, 0, 0, 0).addTo(end);
}

// Weather forecasts are given as timeseries so create an object for that
TimeSeries ts = new TimeSeries();
// set list type
ts.setList(new OrderedList());
// Create a temporal context that defines the start and end
ts.setTemporalContext(start, end);
```

Now, generate some data. In this example it is just random data. In a real-life example you'd fetch this e.g. from a database.

```
 // 1 hour timestep between data values
 Duration timeStep = Factory.createDuration(0, 0, 0, 1, 0, 0);
 Random rnd = new Random();
 Date currentDate = (Date)start.clone();
 while ( currentDate.before(end) ) {
  // Each item in the timeseries is an Evaluation which encodes
  // the values inside it as ValueObjects.
  Evaluation e = new Evaluation();
  int solarAccessValue = 0;
  if ( solarAccess ) {
   solarAccessValue = rnd.nextInt(101);
   ValueObject v = new ValueObject();
   v.setQuantity(RESOURCE.SOLARACCESS);
   v.setUnit(RESOURCE.PERCENT);
   v.setValue(new Variant(solarAccessValue));
   e.addValueObject(v);
  }
  if ( cloudiness ) {
   ValueObject v = new ValueObject();
   v.setQuantity(RESOURCE.CLOUDINESS);
   v.setUnit(RESOURCE.PERCENT);
```

```
      v.setValue(new Variant(rnd.nextInt(101-solarAccessValue)));
      e.addValueObject(v);
     }
     if ( temperature ) {
      ValueObject v = new ValueObject();
      v.setQuantity(RESOURCE.TEMPERATURE);
      v.setUnit(RESOURCE.DEGREECELSIUS);
      // random number from 20 to 25
      v.setValue(new Variant(20 + rnd.nextInt(5) + rnd.nextDouble()));
      // if need to convert from celsius to fahrenheit
      if ( !temperatureUnit.equals(RESOURCE.DEGREECELSIUS) ) {
       v = Tools.convertUnit(v, temperatureUnit);
      }
      e.addValueObject(v);
     }
     e.setInstant((Date)currentDate.clone());
     timeStep.addTo(currentDate);
     ts.addListItem(e);
    }
    // put the timeserie into the output
    output.addTimeSerie(ts);
    activity.addOutput(output);
    resp.addActivity(activity);

    // now we have a response object. Let the Tools again convert it into RDF
    ImmutablePair<String, String> responseString =
  Tools.serializeResponse(resp, serialization);
    response.setContentType(responseString.getRight());
    // And send it....
    out.println(responseString.getLeft());
    return;
   } else {
    System.out.println("Invalid x-seas-method header value: " + method);

  out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
      "Invalid x-seas-method header value: " + method),
      serialization));
    return;
   }
  } else {
   System.out.println("Incorrect URL");

  out.println(Tools.toString(Factory.createInvalidRequestErrorResponse(myIdentity,
     "Incorrect URL"),
     serialization));
   return;
  }
 }
```

11.2. An example registration of a service API

This example shows how to register a description ("recipe") of an API to the SEAS search registry. The example is made for the weather service example in step 1 but can easily be modified to suit other purposes.

This example is coded in Java and can run within a servlet engine or simply as a standalone Java program. For examples in other languages, please visit the documentation section of the developer portal.

```
// First, create a service description
// myIdentity should carry the URI of your service
// for identification and linking purposes.
Service service = new Service(myIdentity);
service.setName("Acme weather service example");

// Tell that this service is available from Espoo, Finland
Address addrs = new Address();
addrs.setCity("Espoo");
addrs.setCountry("Finland");
service.setAddress(addrs);

// Activities are the core of services. Each service should
// perform at least one activity. So create one.
Activity activity = new Activity();
service.addCapability(activity);

// Create input description
Input input = new Input();
activity.addInput(input);

// Now, tell the registry what kind of inputs our API wants
// to receive. Start with interval, i.e. the start and end
// dates of our service, placed inside a TemporalContext.
// input may have seas:temporalContext property with exactly one value of
 type seas:TemporalContext
input.hasOptionalProperty(1, RESOURCE.TEMPORALCONTEXT,
 PROPERTY.TEMPORALCONTEXT);
TemporalContext tcx = new TemporalContext();
// temporal context may have seas:start and/or seas:end properties with
 exactly one value of type xsd:dateTime each
tcx.hasOptionalProperty(1, DATATYPE.DATETIME, PROPERTY.START, PROPERTY.END);
input.setTemporalContext(tcx);

// We also want to know the coordinates to where the forecast
// should be provided.
// input has to have seas:parameter property with exactly one value of type
 seas:Parameter
input.hasRequiredProperty(1, RESOURCE.PARAMETER, PROPERTY.PARAMETER);
Coordinates coordinates = new Coordinates();
// coordinates has to have exactly one latitude and one longitude
coordinates.hasRequiredProperty(1, (String)null, PROPERTY.LATITUDE,
 PROPERTY.LONGITUDE);
input.addParameter("coordinates", new Variant(coordinates));

// Finally, our service wants to know the output values it
// puts into the response.
ValueObject solarAccess = new ValueObject();
solarAccess.setQuantity(RESOURCE.SOLARACCESS);
solarAccess.setUnit(RESOURCE.PERCENT);
input.addOutputValue(solarAccess);
ValueObject cloudiness = new ValueObject();
cloudiness.setQuantity(RESOURCE.CLOUDINESS);
cloudiness.setUnit(RESOURCE.PERCENT);
input.addOutputValue(cloudiness);
ValueObject temperatureCelsius = new ValueObject();
```

```
temperatureCelsius.setQuantity(RESOURCE.TEMPERATURE);
temperatureCelsius.setUnit(RESOURCE.DEGREECELSIUS);
input.addOutputValue(temperatureCelsius);
ValueObject temperatureFahrenheit = new ValueObject();
temperatureFahrenheit.setQuantity(RESOURCE.TEMPERATURE);
temperatureFahrenheit.setUnit(RESOURCE.DEGREEFAHRENHEIT);
input.addOutputValue(temperatureFahrenheit);

// You can also describe the output weather data
Output output = new Output();
output.addCategory(RESOURCE.WEATHER);
output.addCategory(RESOURCE.FORECAST);
output.addTimeSerie(new TimeSeries(RESOURCE.TEMPERATURE,
 RESOURCE.DEGREECELSIUS));
output.addTimeSerie(new TimeSeries(RESOURCE.TEMPERATURE,
 RESOURCE.DEGREEFAHRENHEIT));
output.addTimeSerie(new TimeSeries(RESOURCE.CLOUDINESS, RESOURCE.PERCENT));
output.addTimeSerie(new TimeSeries(RESOURCE.SOLARACCESS, RESOURCE.PERCENT));
activity.addOutput(output);

// Then describe the interface description for the service
// api, i.e., where the service is listening for incoming requests
InterfaceAddress ia = new InterfaceAddress();
ia.setScheme("http");
ia.setHost("127.0.0.1");
ia.setPath("TimeSeriesResponder/seas/v1.0e1.0/access/");
ia.setPort(8080);
ia.addContentType(SERIALIZATION.toContentType(SERIALIZATION.TURTLE));
ia.addContentType(SERIALIZATION.toContentType(SERIALIZATION.JSON_LD));
activity.setInterface(ia);

// Once we have a Service object, we can pass it to
// the registration agent. It will handle the rest automatically.
RegistrationAgent agent = new RegistrationAgent(registratorIdentity);
agent.addEntity(service);
agent.registrate(this);
```

11.3. An example client for fetching weather information

This is the client counterparty to the weather service server API. It calls the service, providing it the input parameters it needs, and parses the results into objects.

```


// My SEAS ID URI. This string should uniquely identify the client that
// makes the call.
String myIdentity = "http://compa.ny/seas/CRequester";

// URI of the service API
String serviceUri = "http://seas.examples.com:8080/seas/v1.0e1.0/access/";

// for how many hours from now we request the forecast
int forecastDurationHours = 2;
```

We are making a request, for that we need an object. The SeasObjects Factories will easily create them. We are interested in the output of an activity. So create an activity as the base container for our request. Finally, we need inputs to that activity. In this case those are coordinates and time interval.

```
try {
 Request request = Factory.createRequest(myIdentity);

 // create activity and place it in the request
 Activity activity = new Activity();
 request.addActivity(activity);

 // create input and place it in the activity
 Input input = new Input();
 activity.addInput(input);

 // add coordinates
 input.addParameter("coordinates", new Variant(new Coordinates(60.123,
 24.123)));

 // add time interval as input
 TemporalContext tcx = new TemporalContext();
 Date now = new Date();
 tcx.setStart(now);
 Calendar c = Calendar.getInstance();
 c.setTime(now);
 c.add(Calendar.HOUR_OF_DAY, forecastDurationHours);
 tcx.setEnd(c.getTime());
 input.setTemporalContext(tcx);
```

Next, tell the request which timeseries quantities and units we want to be returned.

```
 // define requested quantities in output values input
 input.addOutputValue(new ValueObject(RESOURCE.SOLARACCESS, RESOURCE.PERCENT,
 (Variant)null));
 input.addOutputValue(new ValueObject(RESOURCE.TEMPERATURE,
 RESOURCE.DEGREEFAHRENHEIT, (Variant)null));
```

Now, send the request. You can do this with any HTTP capable client. For convenience, the SeasObjects library contains an HttpClient that automatically handles objects.

```
 HttpClient client = new HttpClient(HttpClient.SEAS_METHOD_REQUEST);
 Response response = client.sendPost(serviceUri, request);
```

OK, now we have a parsed response. In a real-life application you would do something with the data. In this example, just print them out.

```
 for ( Activity a : response.getActivities() ) {
  for ( Output o : a.getOutputs() ) {
   for ( TimeSeries tss : o.getTimeSeries() ) {
```

```
    for ( Object e : tss.getList().getItems() ) {
     if ( ((Evaluation)e).hasInstant() ) {
      System.out.println("Timestamp: " +
 Tools.dateToString(((Evaluation)e).getInstant()) );
     } else {
      System.out.println("New item without timestamp: ");
     }
     for ( ValueObject vo : ((Evaluation)e).getValueObjects() ) {
      if ( vo.hasQuantity() ) {
       System.out.println("  Quantity: " + vo.getQuantity());
      }
      if ( vo.hasUnit() ) {
       System.out.println("  Unit: " + vo.getUnit());
      }
      if ( vo.hasValue() ) {
       System.out.println("  Value: " + vo.getValue().toString());
      }
     }
    }
   }
  }
 }

} catch ( Exception e ) {
 System.err.println("Failed to fetch weather forecast");
}
```

11.4. An example search for services

When we are interested in finding services or other entities to connect to, the SEAS registry will find these for us. This example, programmed in Java will search all services that are registered to be located in a given rectangular area. All search features are detailed in the SEAS Registration Service document.

The search will be performed by the search agent. All it needs is parameters for the search, it will do the semantic parsing and serialization automatically.

```
SearchAgent agent = new SearchAgent("http://example.code.com/caller_id");
```

We want to find entities that are of class seas:Service

```
agent.ofType(RESOURCE.SERVICE);
```

We're not interested in offline services. The service must have registered itself at most three days ago.

```
agent.daysOldData(3);
```

Set the geographical area for the search

```
agent.rectangleSearchArea(new Coordinates(60.1325, 24.7794), new
 Coordinates(60.2179, 24.9511));
```

... and execute.

```
ArrayList<Entity> entities = agent.search();
```