An easy, succinct Azure Java SDK API design using modern design patterns and simplified structure

*By Martin Sawicki; Last update: September 15, 2015*

# Introduction

This is an experimental prototype of a new approach to Azure Java API design enabling user client code that is:

1. Easier to read and write
2. More succinct
3. At least on par, if not better, than the competition

# Why

This proposal is the result of research into the challenges of the current Azure SDK API surface as understood from:

* Customer feedback
* Personal experiences writing PoCs and samples on Azure
* Competitive comparison

# Design strategy

The design approach is largely inspired by (but not necessarily strictly adhering to):

* Modern, increasingly popular Java API design patterns, specifically: the ***fluent interface*** and the ***builder******pattern*** (some great and succinct background for the two is [here](http://stackoverflow.com/a/17937946))
* Other clouds’ API design, many of which seem to be roughly following these patterns as well
* Additional twists on those design patterns for better code writing experience
* Reasonable default values for required parameters, whenever possible, to simplify “getting started from scratch” experience.
* A much flatter object hierarchy with more direct access to object settings and fewer layers of indirection

# Example: Provisioning a VM

The following 3 code samples all achieve the same end result: **starting a Linux VM instance** from scratch, in an empty account, using the minimum amount of code. This scenario does a particularly dramatic job of highlighting the difference among the **current** **Azure SDK** API for Java, the **AWS** SDK for Java and the **proposed** **“shortcuts”** API layer. It is illustrative of both the existing challenge in the SDK as well as the proposed solution:

|  |  |  |
| --- | --- | --- |
| Current Azure SDK | AWS SDK | Proposed API design for Azure |
| // Get authenticated client  Configuration config = PublishSettingsLoader.*createManagementConfiguration*(  publishSettingsFilePath, subscriptionID);  ComputeManagementClient computeClient =  ComputeManagementService.*create*(config);  // Create cloud service  **final** HostedServiceCreateParameters serviceParams =  **new** HostedServiceCreateParameters();  serviceParams.setLocation(region);  serviceParams.setServiceName(serviceName);  computeClient.getHostedServicesOperations()  .create(serviceParams);    // Create SSH endpoint  InputEndpoint sshPort = **new** InputEndpoint();  sshPort.setName("ssh");  sshPort.setProtocol("tcp");  sshPort.setLocalPort(22);  sshPort.setPort(22);  ArrayList<InputEndpoint> endpoints =  **new** ArrayList<InputEndpoint>();  endpoints.add(sshPort);  // Create net configuration set  ConfigurationSet netConfigSet = **new** ConfigurationSet();  netConfigSet.setConfigurationSetType(  ConfigurationSetTypes.***NETWORKCONFIGURATION***);  netConfigSet.setInputEndpoints(endpoints);  // Prepare configuration sets collection  ArrayList<ConfigurationSet> configs = **new** ArrayList<ConfigurationSet>();  configs.add(netConfigSet);  // Configure login (assume the OS is Linux)  ConfigurationSet osConfigSet = **new** ConfigurationSet();  osConfigSet.setConfigurationSetType(  ConfigurationSetTypes.***LINUXPROVISIONINGCONFIGURATION***);  osConfigSet.setAdminPassword(password);  osConfigSet.setUserPassword(password);  osConfigSet.setDisableSshPasswordAuthentication(**false**);  osConfigSet.setUserName(userName);  osConfigSet.setComputerName(serviceName);  osConfigSet.setEnableAutomaticUpdates(**true**);  osConfigSet.setHostName(serviceName);  configs.add(osConfigSet);  // Create storage account  StorageManagementClient storageClient =  StorageManagementService.*create*(config);  **final** StorageAccountCreateParameters storageParams =  **new** StorageAccountCreateParameters();  storageParams.setDescription(storageName);  storageParams.setLocation(region);  storageParams.setName(storageName);  storageParams.setAccountType(storageType);  storageClient.getStorageAccountsOperations()  .create(storageParams);    // Determine URL of VHD blob to use  StorageAccountGetResponse account = storageClient.getStorageAccountsOperations().get(storageName);  ArrayList<URI> uris =  account.getStorageAccount().getProperties().getEndpoints();  String vhdURL = **null**;  **for**(URI uri : uris) {  **if**(uri.toASCIIString().contains(".blob.")) {  vhdURL = uri.toASCIIString();  **break**;  }  }  vhdURL += "vhd/vhd" + System.*currentTimeMillis*() + ".vhd";  // Creates OS virtual disk  OSVirtualHardDisk osDisk = **new** OSVirtualHardDisk();  osDisk.setMediaLink(**new** URI(vhdURL));  osDisk.setSourceImageName(imageID);  // Prepare role definition  Role role = **new** Role();  role.setConfigurationSets(configs);  role.setProvisionGuestAgent(**true**);  role.setRoleName(serviceName);  role.setRoleSize(vmSize);  role.setRoleType(VirtualMachineRoleType.***PersistentVMRole***.toString());  role.setOSVirtualHardDisk(osDisk);  ArrayList<Role> roles = **new** ArrayList<Role>();  roles.add(role);  // Run VM  **final** VirtualMachineCreateDeploymentParameters vmCreateParams =  **new** VirtualMachineCreateDeploymentParameters();  vmCreateParams.setRoles(roles);  vmCreateParams.setDeploymentSlot(DeploymentSlot.***Production***);  vmCreateParams.setLabel(serviceName);  vmCreateParams.setName(serviceName);  computeClient.getVirtualMachinesOperations()  .createDeployment(serviceName, vmCreateParams); | // Get authenticated client  AmazonEC2Client client = **new** AmazonEC2Client()  .withRegion(Regions.***US\_WEST\_2***);    // Create a new security group  **final** String name = "mySecGroup";  **final** String description = name;  client.createSecurityGroup(  **new** CreateSecurityGroupRequest(  name, description));  // Create SSH endpoint  IpPermission ipPermission = **new** IpPermission()  .withIpRanges("78.131.140.3/32")  .withIpProtocol("tcp")  .withFromPort(22)  .withToPort(22);    client.authorizeSecurityGroupIngress(**new** AuthorizeSecurityGroupIngressRequest(  name, Arrays.*asList*(ipPermission)));  // Configure login (assume the OS is Linux)  **final** String keyPairName = "myKeyPair";  CreateKeyPairResult keyPairResult = client.createKeyPair(  **new** CreateKeyPairRequest(keyPairName));  KeyPair keyPair = keyPairResult.getKeyPair();  System.***out***.println("Key private: "  + keyPair.getKeyMaterial());    // Run VM  **final** String imageID = "ami-13471c23";  RunInstancesRequest runInstanceRequest = **new** RunInstancesRequest(imageID, 1, 1)  .withInstanceType(InstanceType.***T1Micro***)  .withKeyName(keyPairName)  .withSecurityGroups(name);  client.runInstances(runInstanceRequest); | // Get authenticated client  **final** Azure azure = **new** Azure(publishSettingsPath, subscriptionId);    // Run VM in a new service, with SSH login set up  String imageID = "b39f27a8b8c64d52b05eac6a62ebad85\_\_Ubuntu-12\_04\_5\_LTS-amd64-server-20150413-en-us-30GB";  azure.virtualMachines.define(vmName)  .withRegion("West US")  .withSize("Small")  .withAdminUsername(adminUsername)  .withAdminPassword(adminPassword)  .withLinuxImage(imageID)  .withTcpEndpoint(22)  .provision(); |

# Design pattern details

API Design patterns are proposed for the following classes of API actions on “top level” cloud entities:

1. **Listing** existing cloud entities
2. **Creating** new cloud entities
3. **Reading** existing cloud entity settings
4. **Updating** existing cloud entity settings
5. **Deleting** existing cloud entities

## Authentication

API operations discussed here require authentication. Today’s SDK requires this to be at least a 2-step process: creating a configuration with the credentials and then instantiating one of several clients depending on the general type of service the code will be interacting with. The proposal is to simplify this process so that, similar to AWS:

* Providing the credentials and creating the client object is one and the same step
* There is only one client that the user needs to be aware of

// Get authenticated client

**final** Azure azure = **new** Azure(publishSettingsPath, subscriptionId);

In the above example, a legacy publish settings file and a subscription is provided. Other, newer Azure authentication methods should naturally be supported as well, such as certificates and AD tokens, just as different constructors.

The client object, once created, should be immutable.

## Top level cloud entities

What is understood by *top level* cloud entities here is a set of cloud entities we can identify that the programmer would likely naturally expect to have direct access to, from the level of the subscription. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| Cloud services | Virtual machines | Regions | VM sizes |
| Virtual networks | Storage accounts | OS images | VM images |
| Etc… *(TBD)* |  |  |  |

Counter examples of objects that are clearly **not** **top level**, but are more naturally thought of as children of other objects include:

|  |  |  |  |
| --- | --- | --- | --- |
| Role endpoints | Blobs in a container | Containers in account | Subnets of a network |
| Etc… *(TBD)* |  |  |  |

Although some of the principles discussed here may or may not apply to objects that are not top level, the proposed design is intended to be applicable to managing primarily top level cloud entities.

The authenticated client would expose access to them via **interfaces** only (no direct access to classes). For example, a VirtualNetworks interface for manipulating virtual networks would be available via azure.**networks**. (*Side note*: since the client object is immutable, they could be exposed as *final fields* in Java, rather than methods, to reduce needless verbosity). Other examples:

|  |  |  |  |
| --- | --- | --- | --- |
| azure.**regions** | azure.**virtualMachines** | azure.**osImages** | azure.**vmImages** |
| azure.**cloudServices** | azure.**sizes** | azure.**networks** | azure.**storageAccounts** |
| Etc… *(TBD)* |  |  |  |

All these top level interfaces would at minimum expose the following methods:

* list()
* get(String name)

Interfaces representing cloud entity families that the user can create, update and delete would additionally expose:

* define(String name)
* update(String name)
* delete(String name)

Also note that, in short, define() and update() here are the starting points of a two-step *builder pattern-*like API design, with the *fluent interface* pattern used for exposing setters on the corresponding \*Definition and \*Updatable sets of interfaces, each eventually also exposing a provision() method (for Definiton) or apply() (for Updatable) to finalize the creation or the update process. This is discussed in greater detail later.

## Listing cloud entities

All top level interfaces representing collections of entities would expose a **list()** method to return a **String array** with their unique names. [*Side note*: these top level interfaces could additionally be iterable, since they represent collections, but I don’t think it’s a must, and there are some arguments against that.]

In some cases, additional list() overloads would be provided for additional filtering. For example, Azure.regions.list() could expose a way to list only those regions that support a given service.

Examples:

|  |  |
| --- | --- |
| List available regions | // List all regions  azure.regions.list()  // List regions supporting IaaS only  azure.regions.list(LocationAvailableServiceNames.***PERSISTENTVMROLE***) |
| List available OS images | azure.osImages.list() |
| List available VM sizes | // List all VM sizes  azure.sizes.list()  // List VM sizes supporting IaaS only  azure.sizes.list(**true**, **false**)  // List VM sizes supporting PaaS only  azure.sizes.list(**false**, **true**) |
| List virtual networks | azure.networks.list() |
| List cloud services | azure.cloudServices.list() |
| List storage accounts | azure.storageAccounts.list() |

## Creating cloud entities

Following a builder pattern-like approach, all top level interfaces supporting the creation of a new entity would expose a **define(String name)** method providing access to the (first) interface that exposes fluent interface-style setters for all the required and optional settings. The (final) interface, representing the object with all the required parameters set, would then expose a **provision()** method that initiates the actual creation of the entity in the cloud. (This method could also return an *Updatable* interface to the object - see the later section on cloud entity updating)

Due to the inherent complexity and the large number of required and optional inputs necessary for a number of “creation” tasks in Azure, this design takes advantage of modern and increasingly popular Java API patterns: the popular *builder pattern* combined with the *fluent interface* as well as some additional enhancements discussed later. Their goal is to optimize client code *readability* and *writeability.*

This proposed approach would be a radical simplification of today’s auto-generated SDK design, where layers upon layers of objects and arrays need to be instantiated and assembled together by the programmer, with a lot of old-style, verbose setters. Creating a VM is a great example of that. Instead, what today consist of dozens of lines of code (as shown earlier) could be expressed as a single, very readable statement shown below:

// Create a VM in a new service, with SSH login set up

azure.virtualMachines.define(vmName)

.withRegion("West US")

.withSize("Small")

.withAdminUsername(adminUsername)

.withAdminPassword(adminPassword)

.withLinuxImage(imageID)

.withTcpEndpoint(22)

.provision();

The general builder pattern-like design here is Define / Customize / Provision:

1. Define - The interface representing top level cloud entities (e.g. VirtualMachines) would expose a **define(String name)** method, requiring a unique String identifier for the entity - a common requirement throughout the Azure management surface
2. Customize - The returned template-like “Definition” interface would expose fluent interface-style setter methods for the coder to provide all the *required* and *optional* settings, for example **withSize(String size)**. To enable convenient command chaining, fluent interface-style setters return the definition object itself, rather than void as has been the traditional and more verbose setter pattern of the past. The “with” prefix naming convention helps identify such setters in auto-complete and distinguish from the old style void-returning setters in Java. This is very similar to AWS’s approach.
3. Provision - A “definition” interface returned by the last *required* setter would expose a **provision()** method to initiate the provisioning process on the cloud side. (*Side note*: the word “provision” is chosen intentionally here to distinguish from the traditional client-side object creation, which is typically represented by “build()” or “create()”.)

Examples:

|  |  |
| --- | --- |
| Create a **Linux VM** in a *new* service deployment | azure.virtualMachines.define(vmName)  .withRegion("West US")  .withSize("Small")  .withAdminUsername(adminUsername)  .withAdminPassword(adminPassword)  .withLinuxImage(imageID)  .withTcpEndpoint(22)  .provision(); |
| Create a **Windows VM** in an *existing* service deployment | azure.virtualMachines.define(vmName)  .withExistingCloudService(serviceName)  .withSize("Small")  .withAdminUsername(adminUsername)  .withAdminPassword(adminPassword)  .withWindowsImage(imageID)  .withTcpEndpoint(3389)  .provision(); |
| Create **virtual network** with 1 default subnet | azure.networks.define(networkName)  .withRegion("West US")  .withCidr("10.0.0.0/29")  .provision(); |
| Create an empty **cloud service** | azure.cloudServices.define(serviceName)  .withRegion("West US")  .provision(); |
| Create a **storage account** | azure.storageAccounts.define(accountName)  .withRegion("West US")  .provision(); |

## Reading cloud entity info

The detailed information about any top level cloud entity accessible to the user would be accessible with a **get(String id)** method, returning an **immutable** interface exposing the information about the entity via simple, often String-returning accessor methods and a mostly flat hierarchy.

[*Side note*: as shown in the examples below, because the object is immutable, Java’s “get” naming convention for accessor methods is not applicable, so the “get” prefix is intentionally skipped.]

Examples:

|  |  |
| --- | --- |
| Read a **virtual network** | Network network = azure.networks.get(name);  System.***out***.println(String.*format*("Virtual network: %s\n"  + "\tRegion: %s\n"  + "\tCIDR: %s\n",  network.name(),  network.region(),  network.cidr()));  // Etc... |
| Read a **cloud service** | CloudService cloudService = azure.cloudServices.get(name);  System.***out***.println(String.*format*("Cloud service: %s\n"  + "\tDescription: %s\n"  + "\tRegion: %s\n",  cloudService.name(),  cloudService.description(),  cloudService.region()));  // Etc... |
| Read a **storage account** | StorageAccount storageAccount = azure.storageAccounts.get(name);  System.***out***.println(String.*format*("Storage account: %s\n"  + "\tLabel: %s\n"  + "\tDescription: %s\n",  storageAccount.name(),  storageAccount.label(),  storageAccount.description()));  // Etc... |
| Read an **OS image** | OSImage osImage = azure.osImages.get(name);  System.***out***.println(String.*format*("Found image: %s\n"  + "\tCategory: %s\n"  + "\tDescription: %s\n"  + "\tFamily: %s\n",  osImage.name(),  osImage.category(),  osImage.description(),  osImage.family()));  // Etc... |

## Updating cloud entities

Just like in the case of creation, a builder pattern-like approach should be followed for updating as well. All top level entity interfaces supporting updates would expose an **update(String name)** method providing access to an *Updatable* interface exposing fluent interface-style setters for all the required and optional settings. As long as all update settings are optional, all of them could be exposed on just that one *Updatable* interface, along with an **apply()** method which initiates the actual update process in the cloud.

|  |  |
| --- | --- |
| Update **cloud service** settings | azure.cloudServices.update(serviceName)  .withDescription("My updated descr.")  .withLabel("My updated label")  .apply(); |
| Update **storage account** settings | azure.storageAccounts.update(accountName)  .withDescription("My updated descr.")  .withLabel("My updated label")  .apply(); |

For convenience, a parameter-less update() method could also be consistently exposed for the updatable cloud entities on the interface returned from get(name) method. Moreover, both the provision() and apply() methods could return the *Updatable* interface itself for that new or updated cloud entity, since presumably if the programmer can create or update an object, they should be able to update it later as well.

## Deleting cloud entities

Deletion is simple. All top level interfaces representing *deletable* cloud entities would expose an void-returning **delete(String name)** method. Additional delete() overloads could be exposed for additional options; for example, when a VM is deleted, an additional boolean could be provided to specify whether the attached disks should be deleted as well.

|  |  |
| --- | --- |
| Delete a **virtual network** | azure.networks.delete(networkName); |
| Delete a **cloud service** | azure.cloudServices.delete(serviceName); |
| Delete a **storage account** | azure.storageAccounts.delete(accountName); |

For convenience, a parameter-less delete() method should also be exposed on the *Updatable* interface returned by apply() and provision().

## Special enhancements to fluent interface

For the define()/provision() pattern described earlier, there is an additional enhancement to the typical fluent interface pattern I am proposing here. It is not obvious from the earlier code sample because it affects only the code-*writing* experience, not the code reading. But the goal, in a nutshell, is to make it obvious to the programmer what the required vs. optional parameters are and **not** to give access to the provision() method *at all* until the programmer has called all the required withFoo() setters.

This can be done by having the “with” setters representing the *required* parameters returning only *partial* interfaces on the underlying *Definition* objects. Each such partial interface would expose only the next required “with”-setter. Only the last required setter would expose an actual *Provisionable* interface with the provision() method and all the remaining *optional* “with” setters. This is the approach I pursued in the proof of concept.

To understand the value this is providing, let’s discuss some of the pros and cons of the conventional fluent interface pattern.

### Advantages of conventional fluent interface

Arguably the biggest advantage of the fluent interface pattern, especially when combined with the builder pattern, is the significant enhancement of code readability and succinctness in entity creation scenarios. The alternative - constructor-based - approach would quickly become an unreadable anti-pattern in Java when a large number of required input parameters are necessary. At the same time, fluent interface makes the code less verbose than it would be in the traditional getter/setter pattern.

### Disadvantages of conventional fluent interface

There is a tradeoff though. In the conventional fluent interface pattern, it is not clear for the code writer how to easily distinguish between *required* vs *optional* settings. While this would have been traditionally resolved by putting all the required parameter settings on the constructor (or the “define” method) and only expose the optional ones via the setters, that would often bring back the anti-pattern of having too many input parameters in a function signature or a constructor.

### Alternative using “partial” interfaces

The proposed variation on the fluent interface pattern, using “partial” interfaces, therefore solves both challenges.

The very first interface returned by define() would expose only the “with” setter for the *first required* parameter. Then, that setter would return the interface that contains only the setter for the *second required* parameter. And so on, until the last *required* setter returns the interface that exposes all the remaining *optional* setters along with the provision() method.

This way the developer would never be able to write code that is invalid or incomplete from the “required” vs. “optional” parameter perspective. The IDE’s auto-complete would provide the needed assistance while the compiler would simply not compile code with such erroneous logic.

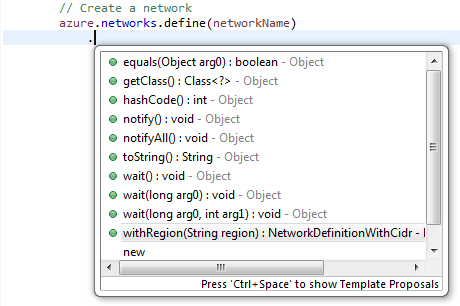
Note that this approach does *not* require multiple class instantiations. The underlying *Definition* object instance can be exactly the same all along, it’s just that since the user does not have direct access to the class anyway, only via interfaces, the interfaces can determine what subset of the underlying object to expose to the programmer, in what circumstances (i.e. as return values from the methods).

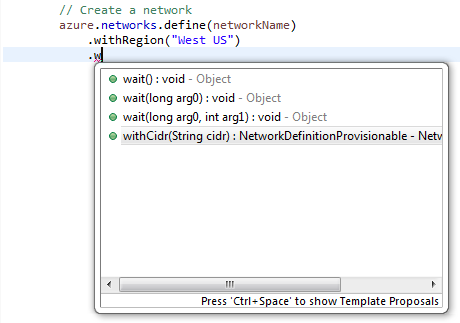
[*Side note*: for that matter, my proof of concept has only one class implementation per type of cloud entity - e.g. VirtualMachine - regardless of whether it is obtained via a get(), or update(), or define(), or provision(). One class can easily implement the entirety of the logic and only the interfaces it exposes dictate what the programmer has access to and under what circumstances.]

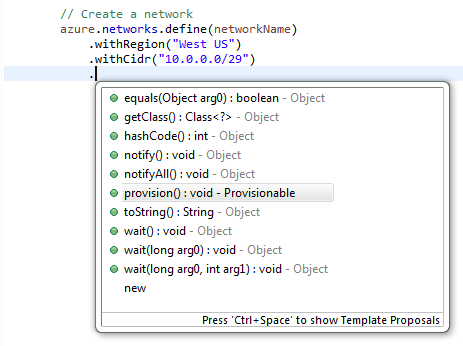
### Example: “Partial interface” variation in action

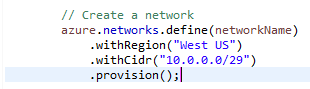
Here’s is the sequence of the Eclipse auto-complete screenshots resulting from this approach. For brevity, the code being written here is for creating a *virtual network*, rather than a VM. VM creation would have required more inputs. But the idea is the same.

Note how the IDE’s auto-complete keeps working with the coder, guiding him step by step through all the required parameter settings and preventing him from calling provision() until the required minimum has been specified. The consistent use of the “with” prefix on the fluent interface setters makes parameter discoverability easier. The resulting code is both easier to read *and write*.









### Effect on mutually exclusive and conditionally required parameters

There is an additional, albeit more subtle, advantage in this “partial interface” tweak to the fluent interface pattern.

It is often the case in the Azure API where two or more required parameters are *mutually exclusive*. For example, in many service-provisioning calls, Azure requires a region *or* an affinity group, but *not both*. Even more complex is the situation where - depending the required input parameter provided by the coder - another one or more inputs become required.

Such conditional parameter inter-dependency can be tricky to expose using traditional approaches. Constructor and method overloads have been used for such design challenges in the past, but as mentioned earlier, they introduce other problems.

However if the setter methods return interfaces that determine the subset of setters to expose next, it becomes much easier to guide the code writer in the right direction even in such more complex scenarios.

For example, to create a cloud service, one must provide either a region or an affinity group, but not both. So the first interface returned by define() could expose just two setters:

* withRegion(String region)
* withAffinityGroup(String affinityGroup)

Then, the interfaces returned by these two would no longer provide access to either, therefore preventing the user from creating such an invalid definition.

This approach makes other, more complex, conditionally-required parameters easier to expose as well.

# Relation to existing Azure SDK

The proposal here is **not** to replace the existing auto-generated API, but to implement this as a “shortcut” layer on top of it, **supplementing** it with a more coder-friendly and succinct API, for at least some key API scenarios (I do not consider it a must-have goal to expose all Azure functionality in this layer). My prototype is implemented on the existing Azure SDK for Java.

Due to its largely auto-generated nature, the current Azure Java API possesses some arguably valuable advantages, some of which may be harder - and perhaps unnecessary - to replicate in this streamlined design. But there are also some significant disadvantages in the current SDK implementation that have proven to be a major deterrent for users.

### Advantages of the current Azure Java API design:

* Implementation **efficiency** for Microsoft
* **Breadth and depth** of Azure functionality coverage (or: matter of prioritization more than technical capabilities)
* **Thinness** of the client-side interop layer implementation (which has some A/C-privileged LCA advantages)
* **Reliability**, no worse than REST for the most part

### Disadvantages of the current API design:

* **Verbose,** at times extremely,
* **Unintuitive**, with unexpected object hierarchies and layers of indirection
* **Undocumentable**, meaning: it is not merely poorly documented today, but it is intrinsically hard to document due it its inherent structure - we wouldn’t be able to create the kind of short “Getting Started” tutorial that AWS has today. Even it was well written – it would still look like an unwieldy behemoth, simply because of the current API design.

[*Side note*: For that matter, my PoC effort here originally started out as an attempt to author a set of extensive tutorials and code samples teaching users how to use the current Azure SDK for Java. The goal of the tutorial was to build a “shortcuts” library. After putting considerable effort into that, I realized that the unwieldiness of the current API, even if explained in detail and with working code samples, would compel many of the tutorial’s readers to just skip the tutorial and not bother with the Azure SDK for Java at all, but just use my “shortcuts” library itself.]

# Prototype

The design proposed here is backed by a working and evolving proof of concept. All the code samples are working test cases. As of this writing, the PoC already largely covers the areas of functionality discussed in this paper and is available for a live demo or for sharing:

<https://marcins.blob.core.windows.net/sdkpoc/AzureAPIJavaPoC.jar>

The prerequisites are:

* **JDK v7** or above (OpenJDK or Oracle)
* Your own **Azure account and a PUBLISHSETTINGS** file for it
* **Azure Libraries for Java v0.7.0** or above, including all its dependencies, must be in the class path. To obtain:
  + Either just add the Azure Library for Java using the Azure Toolkit for Eclipse to your Java project (easiest)
  + Or download the “all-in-one” package from: <http://dl.msopentech.com/lib/PackageForWindowsAzureLibrariesForJava.html> and add to your project’s build path external JAR references to the non-source JAR files (i.e. not the ones that say “sources” in the file name