JsonLite

Overview

`JsonLite` is a lightweight JSON class designed to exclude keys from the network payload and support deltas to reduce the overall network traffic and the memory footprint and disk space of the underlying data store. `JsonLite` excludes keys from the payload by predefining them in the form of `enum` (and `String`) constants and deploying them as part of the application binary classes. The `enum` key classes are automatically versioned and generated using the provided IDE plug-in, ensuring full compatibility and coexistence with other versions.

In addition to the code generator, `JsonLite` includes the following features while maintaining the same level of self-describing messaging of JSON.

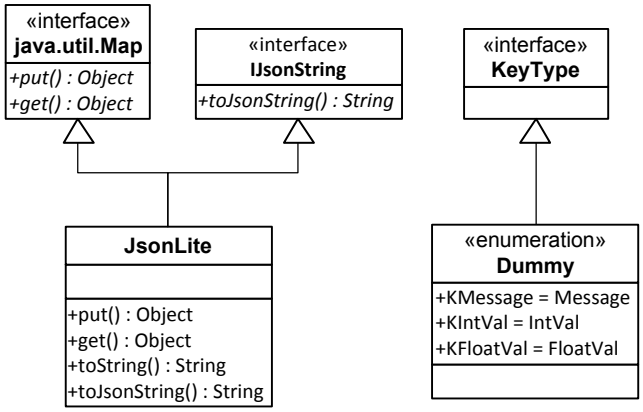
- `JsonLite` is JSON. It is fully JSON compliant.
- `JsonLite` extends JSON to support additional data types. It supports all primitives, all primitive arrays, `Date`, `String`, `String` array, `Map`, `Collection`, `BigDecimal`, `BigInteger`, `Map`, and Java Bean.
- `JsonLite` implements `java.util.Map`.
- `JsonLite`is in general significantly lighter than `HashMap`, JSON, and POJO. Its wire format is compact and does not require class reflection.
- `JsonLite` is faster than `HashMap`, JSON, and POJO. Its smaller payload size means it is serialized and delivered faster.
- `JsonLite` lookup is faster than `HashMap` and JSON. `JsonLite` keeps values internally indexed in an array for faster access.
- `JsonLite` fully supports delta propagation and store.
- `JsonLite` fully supports *selective key inflation* (SKI). With SKI, the underlying `JsonLite` mechanism inflates only the values that are accessed by the application. The rest of the values are kept deflated until they are accessed. This reduces the memory footprint and eliminates the unnecessary latency overhead introduced by the serialization and deserialization operations.
- `JsonLite` fully supports the underlying data grid query service such as GemFire's OQL.
- `JsonLite` is fully integrated with the key class versioning mechanism, which enables multiple versions of `JsonLite` key sets to coexist in the data grid. All versioned key classes are fully forward and backward compatible.
- `JsonLite` key classes are universally unique across space and time, eliminating potential name conflicts.
- `JsonLite` is language neutral.

Lighter and Faster

`JsonLite`, in general, is significantly lighter than `HashMap` and JSON in terms of both size and speed. The size of a typical serialized `JsonLite` object is considerably smaller than the counterpart `HashMap` and JSON object. Enum `get()` calls are faster than `HashMap`'s `get()` because the values are indexed in an array, circumventing the more expensive hash lookup operation.

Map with enum KeyType Keys

`JsonLite` implements `Map` and therefore has the same `Map` methods and behaves exactly like `Map`. Unlike `HashMap` and JSON, `JsonLite` object is restricted to a fixed set of predefined keys in an `enum` class that implements the interface `KeyType`. This restriction effectively makes `JsonLite` lighter, faster, and more acquiescent than `HashMap` and JSON. It removes the keys from the wire format and provides a valid key list for strict validation of allowed keys and types.



Code Generator

Editing keys, although it can be done manually, is done via the provided IDE plug-in which automatically generates a new version of the `enum` class. The built-in versioning mechanism allows the new versioned `enum` class to be deployed to the servers and clients during runtime without the requirement of restarting them. The servers automatically load the new versioned class making it immediately available to the application along with the previous versions.

String Keys

In addition to the `enum` keys, `JsonLite` also supports `String` keys. `String` keys are costlier than `enum` keys but comparable to `HashMap` and JSON in terms of the `put` and `get` speeds. One of the benefits of using `String` keys is the flexibility of executing ad hoc queries. `JsonLite` is fully compliant with the underlying data grid query service such as the GemFire query service, making it ideal for object-relational mapping.

Standard JSON Types

The standard JSON types are *null, boolean, string, number, object, and array*. Strictly out of the box,JSON is capable of supporting only the following types as shown by the JSON reference implementation of *json.org*.

```
null, boolean, int, long, double, JSONArray, JSONObject
```

JsonLite Types

`JsonLite`lifts the JSON type limitations by embedding metadata called the *header* in the JSON string representation to include support foradditional nativeand custom data types. The following is the complete list of the `JsonLite` supported data types.

`null`, all primitive types, `String`, all primitive array types, `Object`, `String[]`, `Object[]`, `java.util.Collection`, `java.util.Map`, `java.util.Date`, `java.math.BigDecimal`, `java.math.BigInteger`, `com.netcrest.pado.data.KeyType`, `JsonLite`, and Java Bean classes.

JsonLite Serialization Rules

From JsonLite to JSON String Representation

- `JsonLite.toString()` returns the compact JSON string representation of the object including the header information. A compact JSON string representation does not include whites space indentations.
- `JsonLite.toString(int indentFactor, boolean isHeader)` can be used to indent JSON string representation and exclude the header information.
- `JsonLite` implements `IJsonString`, which has `toJsonString()`.
- `JsonLite.toJsonString()` is equivalent to `JsonLite.toString()`.

JSON String Representation to JsonLite

Header

String representation can optionally add the header information to provide data types. The header is identified by "__h" (two underscore characters followed by 'h'). The header is an object of the following format:

```
"__h": {
   "c":"<type notation><class name>"
   "u":"<uuid most significant bits>:<uuid least significant
bits>"
   "t":[<comma separated list of array element types in quotes>]
   "d":[<comma separated list of array element values in quotes>]
}
```


where

"__h" is the optional header member. If it is not specified then the default (or standard) JSON object is created. ***The current implementation of JsonLite distributed by Netcrest requires the header member must be defined at the beginning of each object body. In other words, it must be defined immediately after '{'.***

"c" specifies the class name. A class may be a `KeyType` enum class or a Java Bean class. The class name must be a fully qualified class name that includes the package name. For `KeyType`, the "u" key can be optionally used in place of the "c" key as described below.

<type notation> is any one of notations defined in the Type Notations section.

<class name>is a fully-qualified class name. It is required only if the type notation is "L" (object), "[L" (object array), or unspecified.

"u" specifies the UUID assigned to a `KeyType`. A `KeyType` UUID uniquely represents a `KeyType` enum class, which may have multiple versions. It may be used in place of the `KeyType` class name defined by the "c" key. It must be obtained from Pado as part of a JSON response or through the Pado key type management facility.

"t" is an array containing element types in quotes. This member is required only if the "c" is of an array or collection type that contains *heterogeneous* element types. It is ignored otherwise.

"d" is an array containing element values. If the "t" array is specified, then this array must be the same size and contain values that match the types defined in the "t" array. If "t" is not specified, then the `JsonLite` parser complies with the standard JSON object types defined by json.org as described in the Standard JSON Types section.

Body

The body of JSON string representation has two distinct types: object and array. Both types must conform to the JSON grammar with the optional `JsonLite` rules described in the subsequent sections.

JSON Object

An object must be enclosed in `{}` and its members must be key/value paired as described in the JSON language specification.

`JsonLite` is JSON Object. `JsonLite` is normally backed by a schema enum class that implements the `KeyType` interface. `KeyType` classes are automatically versioned and generated by the Pado IDE code generator plug-in. If `KeyType` is not provided then `JsonLite` works as a standard JSON object that supports only the JSON types specified in the Standard

JSON Types section. Unlike JSON, `JsonLite` supports all of the Java primitive types plus more as described in the JsonLite Types section. In addition to its built-in support for `Date`, `BigDecimal`, and `Collection` types, it also supports the `Object` type that conforms to the Java Bean specification, which states a Java Bean class must provide the default no-arg constructor, and setter and getter methods.

JsonLite

In order to create a `JsonLite` object, the header must be included in the string representation. The header information specifies the `KeyType` class that provides the object schema information needed by the `JsonLite` parser to properly construct the `JsonLite` object.

The example below specifies the `TestKey` enum class, which provides member data type information.

```
{
   "__h": {
      "c":" jsonlite.examples.model.TestKey"
   }
   "int":1,
   "double":2.40,
   "string":"test",
   "int[]":[1, 2, 3],
   "double[]":[1.1, 2.1, 3.1],
   "string[]":["abc", "def", "ghi"],
   "array1":[1, 1.1, "abc"],
   "collection1": {
      "__h": {
         "c":"List",
         "t":["I", "D", "T"],
      }
      "_d":[1, 1.1, "abc"]
   }
   "array2": {
      "__h": {
         "c":"[LDate",
      }
      "__d":["2001-07-04T12:08:56.235Z", "2001-07-
04T12:08:56.235Z", "2001-07-04T12:08:56.235Z"]
   }
   "collection2": {
      "__h": {
         "c":"List<Date>"
      }
      "__d":["2001-07-04T12:08:56.235Z", "2001-07-
04T12:08:56.235Z", "2001-07-04T12:08:56.235Z"]
   }
   {
      "__h": {
```



```
         "c": "jsonlite.examples.model.Foo"
      },
      "x": 1,
      "y": 10.1,
      "z": "Foo Test"
   }
}
```

Java Object

If the header specifies a class other than `KeyType` then it is interpreted as a concrete class that loosely conforms to the Java Bean specification. It must supply the default no-arg constructor and setter methods that match the JSON member names. If the object is to be stored in Pado, its class must also implement `java.io.Serializable`. The following is an example Java Bean class.

```
package jsonlite.examples.model;

public class Foo implements java.io.Serializable
{
   int x;
   double y;
   String z;

   public Foo() {}

   public void setX(int x) {
      this.x = x;
   }

   public int getX() {
      return this.x;
   }

   public void setY(double y) {
      this.y = y;
   }

   public double getY() {
      return this.y;
   }

   public void setZ(String z) {
      this.z = z;
   }

   public String getZ() {
      returnthis.z;
   }
}
```


JSON Array

A JSON array must be enclosed in `[]` and its elements must be comma separated as described in the JSON language specification. Unlike JSON, `JsonLite` does not support stand-alone JSON arrays. Instead, all data must be part of `JsonLite` including arrays. In other words, to create an array, it must be mapped by a key in `JsonLite`.

An array comes in three (3) flavors: typed array, `Object[]`, `Collection`. A typed array contains objects of a single homogenous type. `Object[]` contains objects of two or more different types (or heterogeneous types). `Collection` may contain homogenous or heterogeneous types. A header-less JSON string representation always converts to `Object[]`.

Object[]

If the header is an object array, i.e., `"c":"[L"`, then the `Object[]` array type is returned. For example,

```
"__h":{
   "c":"[L",
   "t":["z", "b", "c", "d", "f", "i", "j", "L", "s", "[L", "T",
Date, List<String>]
   "d": ["true", 1, \u2f3a, 1.1, 2.0, 1, 1, {}, 1, [1,2,3],
"string", ""2001-07-04T12:08:56.235Z", { "__h": { "c": "List<T>",
"d": ["str1", "str2"] } } ]

}
```

is equivalent to

```
Object[] { boolean, byte, char, double, float, int, long, Object,
short, Object[], String, java.util.Date,
java.util.ArrayList<String>] }
```

The body must be keyed by `"d"`. For example,

```
"d": [
   "true", 1, \u2f3a, 1.1, 2.0, 1, 1, {}, 1, [1, 2, 3], "string",
   "2001-07-04T12:08:56.235Z",
   {
      "__h": {
         "c": "List<T>",
      "d": [
         "str1",
         "str2"
         ]
      }
   }
]
```


If the header and body do not match then the parser throws an exception.

If the string representation is a standard JSON array, i.e., it is enclosed in `[]`, then `Object[]` is created with the standard JSON supported types.

Collection

JSON string representation can also be converted to Java collection objects by specifying one of the following types in the header:

- `"L<collection class name>"`
  - Example: `"Ljava.util.ArrayList"` or `"Ljava/util/ArrayList"`
- `"List"` – Same as `"Ljava.util.ArrayList"`

Type Notations

`JsonLite` follows the Java internal type notations and naming conventions with additions of several commonly used type notations.

Type	Notation
boolean	z
byte	b
char	c
double	d
float	f
int	i
long	j
short	s
Boolean	Z
Byte	B
Character	C
Double	D
Float	F
Integer	I
Long	J
Short	S
void	V
Object	L
String	T
<array>	[
java.util.Date	Date
java.math.BigDecimal	BD
java.math.BigInteger	BI
java.util.ArrayList	List
java.util.HashMap	Map

Collection Classes and Generic Type

`JsonLite` supports all concrete classes that implement `java.util.Collection`. These classes must also implement `java.io.Serializable` only if they were to be stored in Pado. `JsonLite` also supports the generic type of `Collection`. The generic type declaration syntax is same as Java. For example, `LinkedList` has the generic type of `java.util.Date` can be defined as shown below. Note that because the Java language does not support generic type reflection, a JsonLite implementation must determine the Java generic type by first iterating the entire collection.

```
{
   "__h": {
   "c":"Ljava.util.LinkedList<Date>"
   "d":["2001-07-04T12:08:56.235Z", "2001-0704T12:08:56.235Z",
"2001-07-04T12:08:56.235Z"]
   }
}
```


Using JsonLite

`JsonLite` implements `Map` providing the same services as a typical `Map` implementation. Using `JsonLite` requires the application creating a `KeyTypeenum` class as follows.

1. Create a `KeyType enum` class using the code generator from IDE.
2. Use the generate `KeyType enum` class to create `JsonLite` objects.

JsonLite API

The `JsonLite`API has two sets of methods: `Map` methods and `JsonLite` specific methods.

Map

`JsonLite` implements all of the `Map` interface methods as shown below. Note that the support for generic is provided only for values. The keys must be `KeyType` or `String`.

```
public JsonLite();
public JsonLite(KeyTypekeyType);
public V put(KeyTypekeyType, V value) throwsInvalidKeyException;
public V put(String key, V value) throwsInvalidKeyException;
public void putAll(Map<? extends String, ? extends V> map);
public V get(KeyTypekeyType);
public V get(Object key);
public V remove(Object key);
public void clear();
public int size();
public boolean isEmpty();
public boolean containsValue(Object value);
public boolean containsKey(Object key);
public Set<String>keySet();
public Collection<V> values();
public Set<Map.Entry<String, V>>entrySet();
```

Constructors

`JsonLite` provides the default constructor but its use is discouraged. The preferred way to create `JsonLite` objects is to use the overloaded constructor `JsonLite(KeyTypekeyType)` to ensure the proper initialization of `JsonLite`. If the application must use the default constructor then it must first call `put()` or `get()` using `KeyType` instead of `String` as key so that `JsonLite` can implicitly initialize itself. This is required because `String` keys are freeform keys that do not provide the key type information needed to initialize `JsonLite`.

put()

`JsonLite` supports only `KeyType` and `String` keys. The `put()` methods throw the runtime exception `InvalieKeyException` if the key is invalid or the value has the wrong type.

putAll()

The `putAll()` method performs a bulk-put operation for copying any `Map` contents into `JsonLite`. If the `Map` object type is non-`JsonLite` or `JsonLite` with a different `KeyType`,then only the valid keys in the form of `String` and the values with the correct types are shallow-copied into `JsonLite`. Non-`String` key types are converted to `String` by invoking the `KeyType.toString()` method.

get()

`JsonLite` supports only `KeyType` and `String` keys. The `get(Object key)` method takes an opaque type but internally uses `key.toString()`.

remove()

The `remove()` method removes the value by setting it `null`. The keys are never removed.

clear()

The `clear()` method clears all of the values by setting *non-null* values to `null`. It also marks dirty the *non-null* values.

size()

The `size()` method returns the count of *non-null* values.

isEmpty()

The `isEmpty()` method returns `true` if there are no values. All `null` values are considered no values.

containsValue()

The `contains Value()` method returns `true` if the specified value exists in the `JsonLite` object. It returns `true` if the specified value is `null` and the `JsonLite` object contains one or more `null` values.

containsKey()

The `contains Key()` method returns `true` if the specified key maps a *non-null* value. It uses `key.toString()` to search the key.

keySet()

The `keySet()` method returns all of the `String` keys that map *non-null* values.

values()

The `values()` method returns all of the *non-null* values.

entrySet()

The `entrySet()` method returns the (string key, value) paired entry set that contains only *non-null* values.

JsonLite Specifics

`JsonLite` provides additional methods as shown below.

```
public Object getId();
public int getKeyTypeVersion();
public KeyTypegetKeyType();
public String getName();
public String getKeyTypeName();
public int getKeyCount();
public boolean isDirty();
public boolean hasDelta();
```

getId()

The `getId()` method returns the key type ID that uniquely identifies the `JsonLite` key type. This method call is equivalent to invoking `getKeyType.getId()`.

getKeyTypeVersion()

The `getKeyTypeVersion()` method returns the key type version. There are one or more key type versions per ID.

getKeyType()

The `getKeyType()` method returns the key type instance used to initialize the `JsonLite` object.

getName()

The `getName()` method returns the simple (short) class name of the key type. It returns `null` if the key type is not defined.

getKeyTypeName()

The `getKeyTypeName()` returns the fully qualified class name of the key type. It returns `null` if the key type is not defined.

getKeyCount()

The `getKeyCount()` method returns the count of all keys defined in the key type.

isDirty()

The `isDirty()` method returns `true` if there have been any changes made to the values.

hasDelta()

The `hasDelta()` method returns `true` if delta propagation is enabled and any value changes were made in the `JsonLite` object. Delta propagation can be enabled when creating the `KeyType` class using the IDE plug-in.

Examples

```
import com.netcrest.pado.data.KeyTypeManager;
import com.netcrest.pado.data.jsonlite.JsonLite;
import jsonlite.examples.model.Dummy;
import jsonlite.examples.model.v.Dummy_v1;
import jsonlite.examples.model.v.Dummy_v2;

. . .

// Register the Dummy key type. This is required only for
// non-Pado apps. KeyType registrations for Pado apps are
// automatically performed.
KeyTypeManager.registerKeyType(Dummy.getKeyType());

// Create a JsonLite object using the latest Dummy version.
// Dummy is equivalent to Dummy_v2 if Dummy_v2 is the
// latest version.
JsonLite jl = new JsonLite(Dummy.getKeyType());

// Put data using the Dummy.KMessage enum constant
jl.put(Dummy.KMessage, "Hello, world.");

// Put data using the string key "Dummy" which is equivalent to
// Dummy.KMessage
jl.put("Message", "Hello, world.");

// Get the value using the Dummy.KMessage enum constant
String message = (String) jl.get(Dummy.KMessage);

// Get the value using the versioned enum class Dummy_v2 which is
// equivalentto Dummy if Dummy_v2 is the latest version.
message = (String) jl.get(Dummy_v2.KMessage);

// Get the value using the previous versioned class Dummy_v1.
message = (String) jl.get(Dummy_v1.KMessage);

// Get the value using the string key "Message".
message = (String) jl.get("Message");
```


Installing the JsonLite Eclipse Plug-in

`JsonLite` is a part of the Pado plug-in which is a standard Eclipse and STS (Spring Tool Suite) plug-in. You can either unzip the distribution in the Eclipse root directory or use the Eclipse *Help* menu to install it from the Pado update site. Once installed, you must restart Eclipse or STS for the plug-in to take into effect.

Generating JsonLite KeyType `enum` Classes

Generating a `KeyType enum` class is simple. Let's walk through an example.

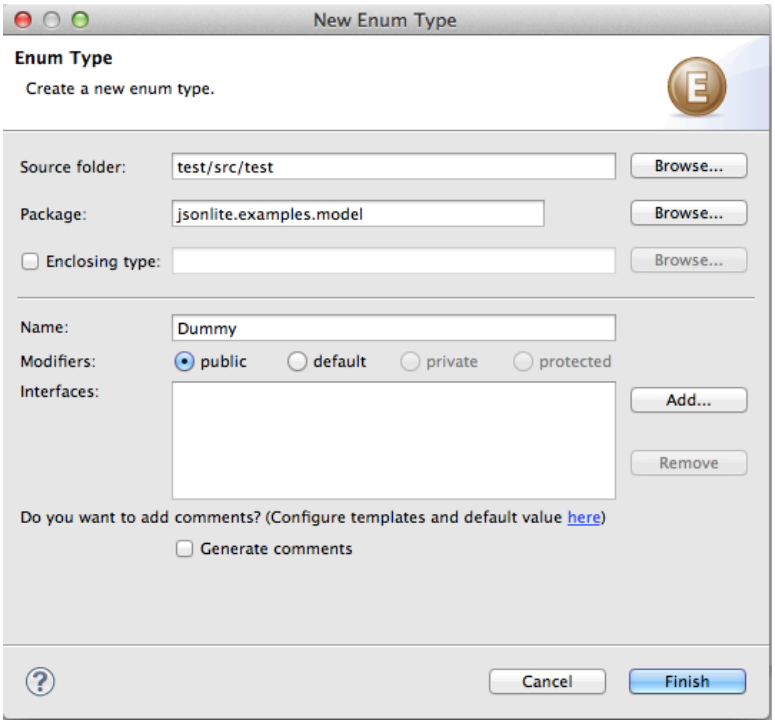
Getting Started

Before using the `JsonLite` plug-in, you must first add the `JsonLite` jar file in your Eclipse/STS project class path. One of the following jar files is required.

- `pado-jsonlite.jar`– The add-on library that contains the `JsonLite` class. Use this jar for non-Pado apps.
- `pado.jar` – The Pado core jar file that also includes `JsonLite` classes. Use this jar for Pado apps.

Creating an Empty `enum` Class

In Eclipse or STS, create an empty `enum` class by selecting the *File/New/Enu*m pull-down menu. In the "New Enum Type" dialog, type in "Dummy" as shown below. Select "Finish" to create the `Dummy` class in the `jsonlite.examples.model` package.

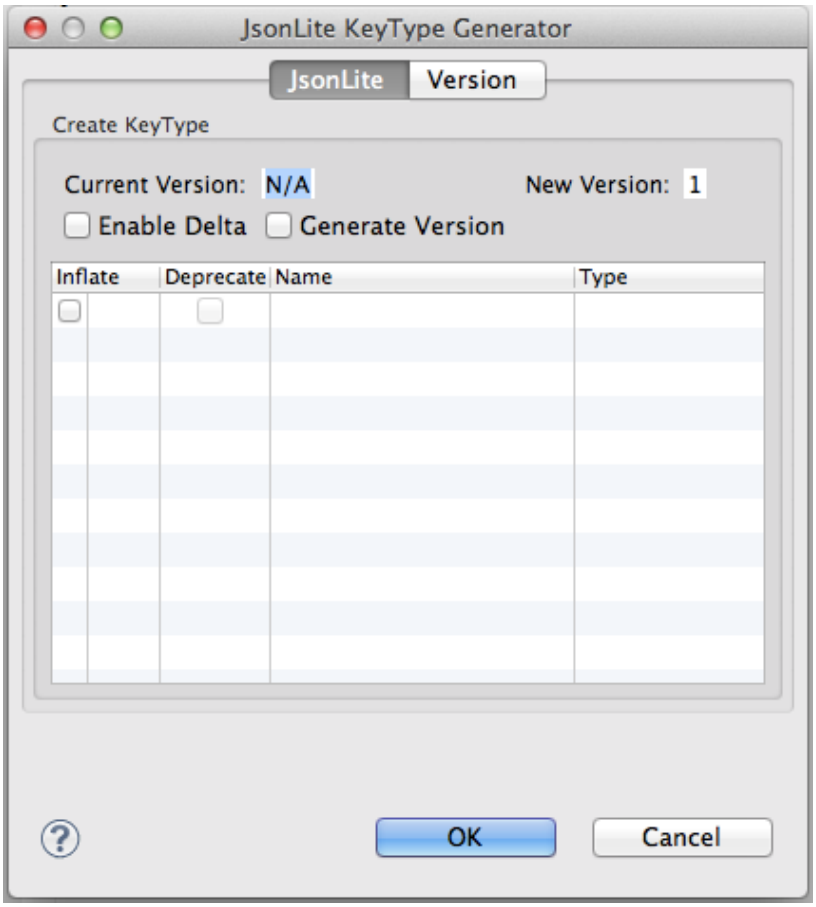


Eclipse creates the following empty `enum` class:

```
package jsonlite.examples.model;

public enum Dummy
{

}
```

Adding keys

Click on the JsonLite icon in the Eclipse/STS toolbar with the `Dummy` class opened in the editor. It launches the "JsonLite Code Generator" dialog as follows.



Let's walk through the options in the dialog.

Enable Delta

Select this option if you want to enable delta propagation. This option must be used with discretion with non-temporal or historical data as the delta propagation overhead may negate

the overall performance gain. For non-temporal or historical data, i.e., data updates, in general, delta propagation should be enabled if the object size is large and/or contains many keys. It is important to note that this limitation does not apply to temporal data. Because temporal data is historical by nature, Pado discretely stores all deltas without relying on the underlying data grid's delta propagation mechanism. Delta propagation is enabled per `KeyType enum` class.

Generate Version

Select this option if you want to generate the versioned class in the "v" sub-package. By default, this is not selected so that you can iteratively modify the `enum` class without generating the versioned class. When you are done modifying the class, you can commit the changes by selecting this option. Note that you can also disable the versioning mechanism by never selecting this option. In that case, since the versioned classes are not available, any changes made in the enum class will not be visible in the server unless the server is restarted.

Inflate

Select this check box to inflate the mapped value. By default, all values are kept deflated (serialized) and inflated only when they are accessed. Having most actively used keys inflated may improve the read performance at the expense of write performance degradation and the memory footprint increase.

Deprecate

Select this check box to deprecate the key from the `enum` class. Deprecated keys are still available to the application. They are removed when a merge is performed.

Name

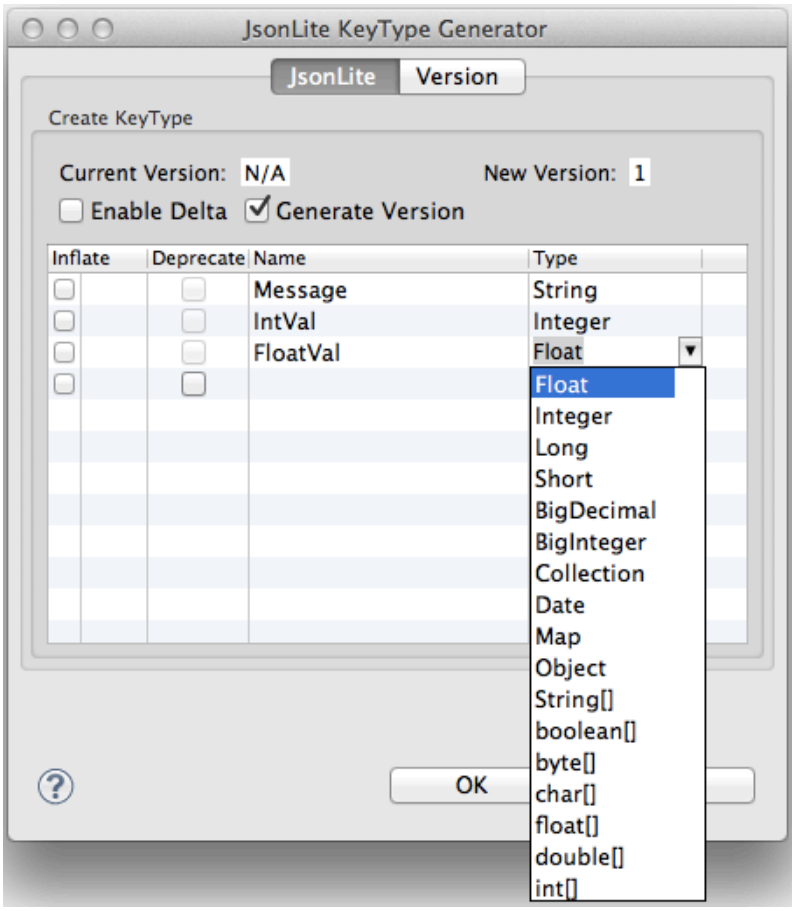
Type in the key name. The key name must be unique and is case sensitive.

Type

Select the combobox from this column to set the key type.

Version 1 – Add Initial Keys

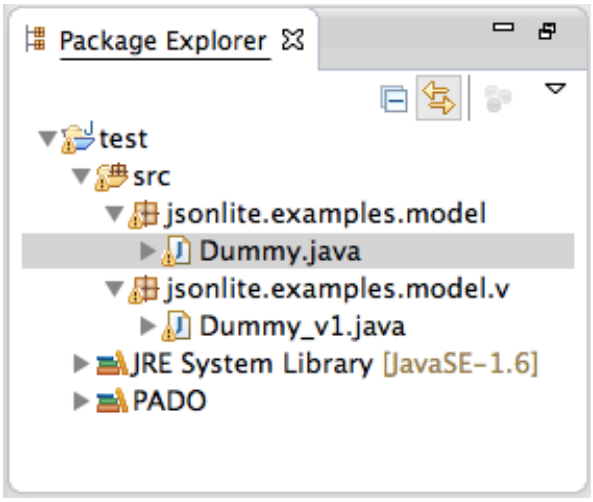
Add key types and select the “Generate Version” checkbox as shown in the figure below and click on the “OK” button to generate the `Dummy` class.



Along with the `KeyType` methods, the above dialog generates the following code in the `Dummy` class (Note that the enum constant names begin with the letter 'K' which stands for "Key".)

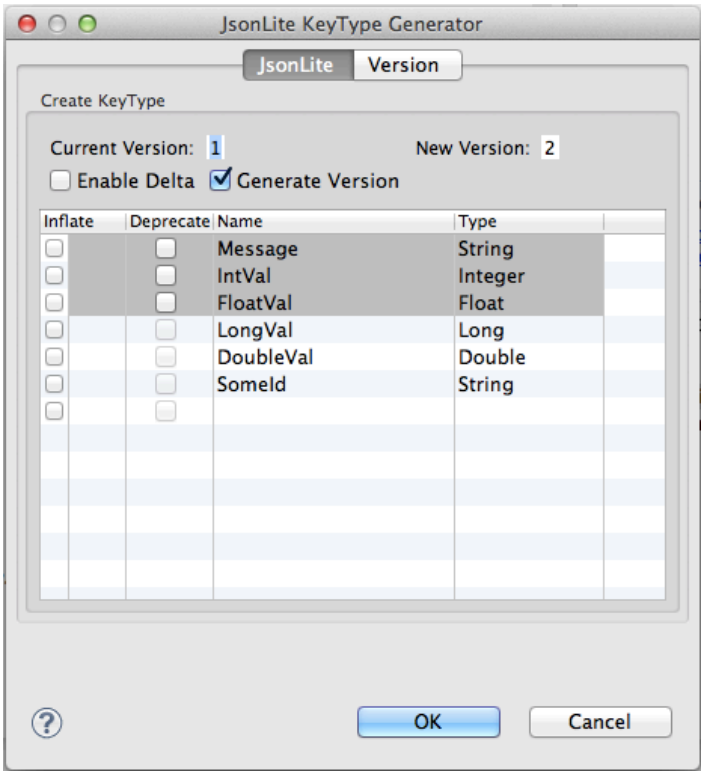
```
/**
 * Message: <b>String</b>
 */
KMessage("Message", String.class, true),
/**
 * IntVal: <b>Integer</b>
 */
KIntVal("IntVal", Integer.class, true),
/**
 * FloatVal: <b>Float</b>
 */
KFloatVal("FloatVal", Float.class, true);
```

It also generates a duplicate class with the name `Dummy_v1` in the `jsonlite.examples.model.v` sub-package as shown below.

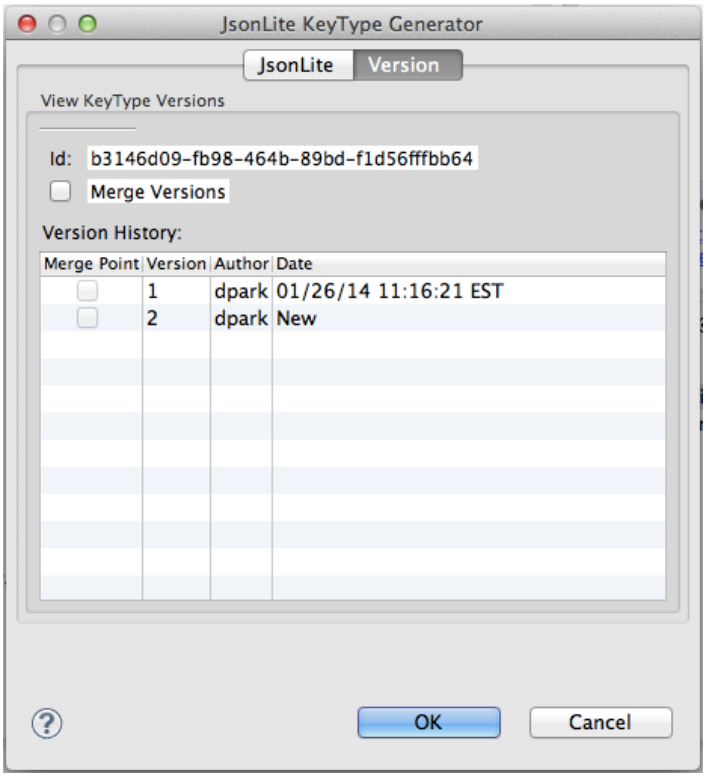


Version 2 – Add New Keys

Click on the JsonLite icon in the toolbar and add three more keys and select the “Generate Version” checkbox as shown below. Note that the Version 1 keys are grayed out. The "Inflate" and "Deprecate" check boxes are editable but the "Name" and "Type" columns for already versioned keys are not editable. Note also that the code generator automatically detects the current version number and assigns the next version number by incrementing it.



Now, select the “Version” tab in the dialog to view the version history as shown below.



The Version tab pane shows the unique ID of the key type class assigned by the code generator when Version 1 was created. This ID cannot be modified as it has been permanently assigned to the `Dummy` class. It is this ID that makes the `Dummy` class universally unique such that no other classes even under the same package can inadvertently overwrite the data.

The Version History table shows the list of the available versions. It basically reflects all of the classes in the "v" sub-package with the `Dummy_v`*<number>* name, where *<number>* is the version number.

The Merge column allows collapsing all of the versions prior to but not including the selected version. Because each `enum` class has a complete set of keys, the code generator does not actually perform merging keys but rather simply deletes the `KeyType enum` classes in the "v" sub-package.

The Author column shows the author who versioned the `KeyType` class.

The Date column shows the time when the class was versioned.

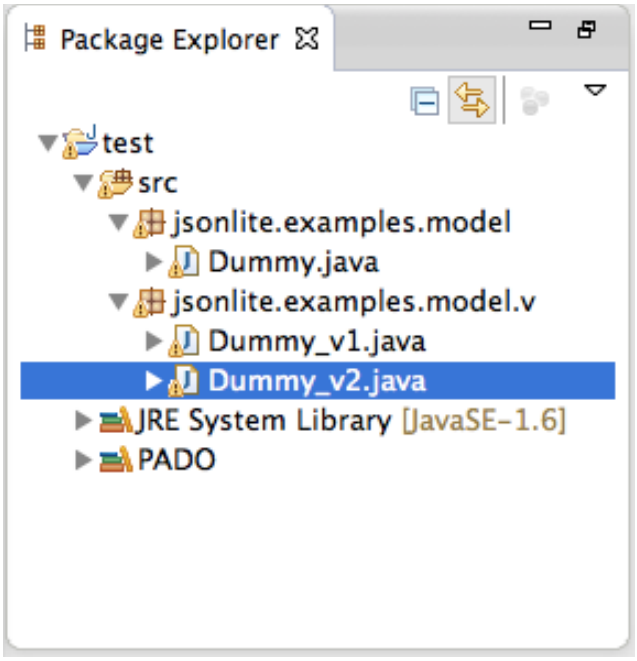
Click on the "OK" button to generate the Version 2 classes. The `Dummy` class now contains the following keys.

```
/**
 * Message: <b>String</b>
 */
KMessage("Message", String.class, true),
```



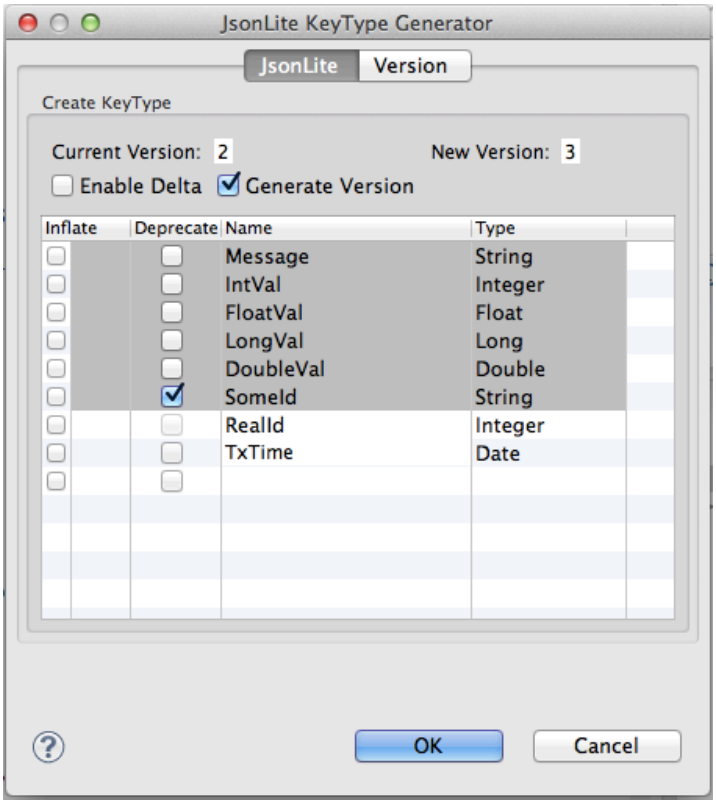
```
/**
 * IntVal: <b>Integer</b>
 */
KIntVal("IntVal", Integer.class, true),
/**
 * FloatVal: <b>Float</b>
 */
KFloatVal("FloatVal", Float.class, true),
/**
 * LongVal: <b>Long</b>
 */
KLongVal("LongVal", Long.class, true),
/**
 * DoubleVal: <b>Double</b>
 */
KDoubleVal("DoubleVal", Double.class, true),
/**
 * SomeId: <b>String</b>
 */
KSomeId("SomeId", String.class, true);
```

It also generates a duplicate class with the name `Dummy_v2` in the `jsonlite.examples.model.v` sub-package as shown below.



Version 3 – Deprecate Keys

Click on the JsonLite icon in the toolbar. Deprecate SomeId, add the new keys, RealId and TxTime, and select the "Generate Version" checkbox in the dialog as shown below.

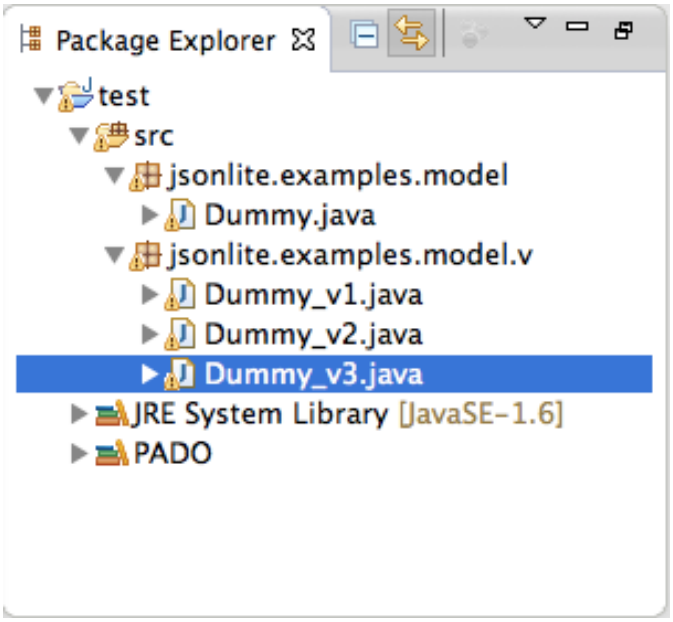


Click on the "OK" button to generate the Version 2 classes. The `Dummy` class now contains the following keys.

```
/**
 * Message: <b>String</b>
 */
KMessage("Message", String.class, true),
/**
 * IntVal: <b>Integer</b>
 */
KIntVal("IntVal", Integer.class, true),
/**
 * FloatVal: <b>Float</b>
 */
KFloatVal("FloatVal", Float.class, true),
/**
 * LongVal: <b>Long</b>
 */
KLongVal("LongVal", Long.class, true),
/**
 * DoubleVal: <b>Double</b>
 */
KDoubleVal("DoubleVal", Double.class, true),
/**
 * SomeId: <b>String</b>
* @deprecated
 */
KSomeId("SomeId", String.class, true);
```

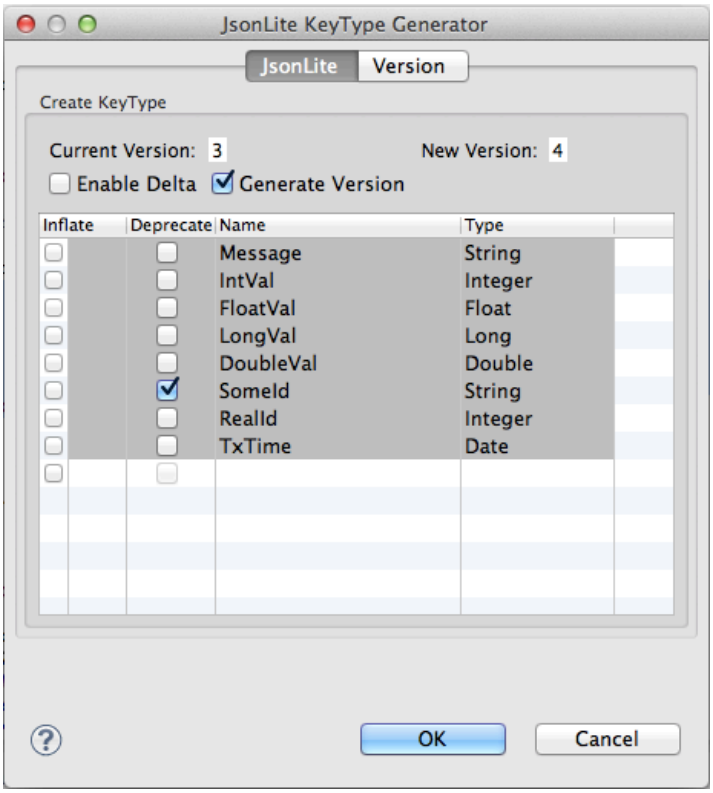
```
/**
 * RealId: <b>Integer</b>
 */
KRealId("RealId", Integer.class, true);

/**
 * TxTime: <b>Date</b>
 */
KTxTime("TxTime", Date.class, true);
```

It also generates a duplicate class with the name `Dummy_v3` in the `jsonlite.examples.model.v` sub-package as shown below.

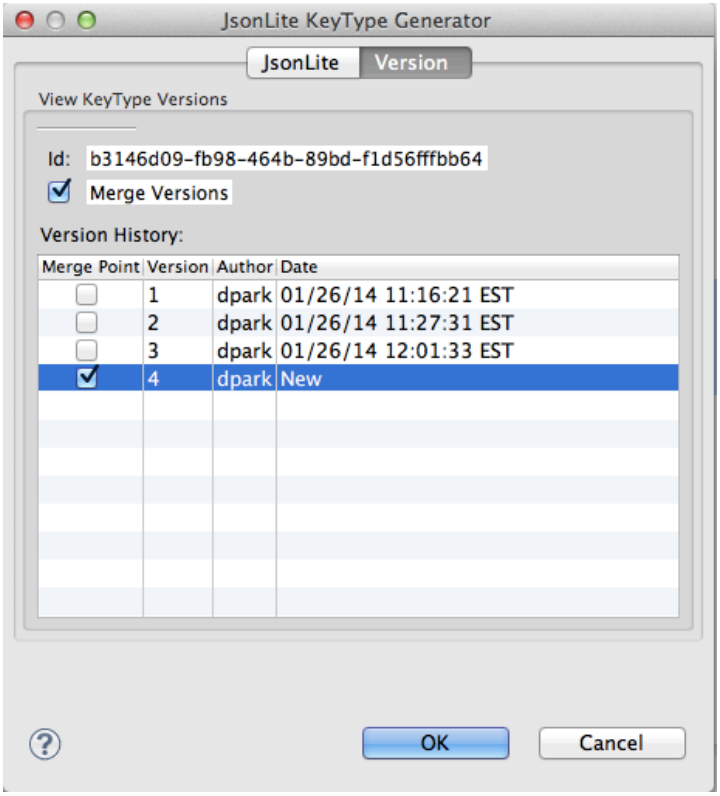


Version 4 – Merge Versions

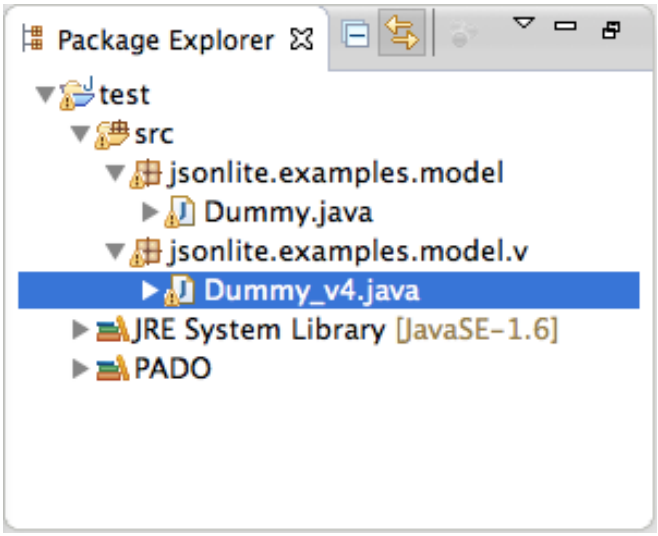
Click on the JsonLite icon in the toolbar. Select the "Generate Version" check box as shown below.



Now select the Version tab, select the "Merge Versions" check box at the top, select the "Merge Point" check box of the Version 4 row, and select the “OK” button as shown below.



The "v" sub-package now contains only the Version 4 class, `Dummy_v4` as shown below. `Dummy_v1`, `Dummy_v2`, and `Dumm_v3` are physically deleted from the file system.



The merged Dummy class now contains only non-deprecated keys as shown below.

```
/**
 * Message: <b>String</b>
 */
KMessage("Message", String.class, true),
/**
 * IntVal: <b>Integer</b>
 */
KIntVal("IntVal", Integer.class, true),
/**
 * FloatVal: <b>Float</b>
 */
KFloatVal("FloatVal", Float.class, true),
/**
 * LongVal: <b>Long</b>
 */
KLongVal("LongVal", Long.class, true),
/**
 * DoubleVal: <b>Double</b>
 */
KDoubleVal("DoubleVal", Double.class, true),
/**
 * RealId: <b>Integer</b>
 */
KRealId("RealId", Integer.class, true);

/**
 * TxTime: <b>Date</b>
 */
KTxTime("TxTime", Date.class, true);
```


Important: *If deprecated keys are involved when merging versions, then the final merged version will have the wire format that is different from the previous versions. This means if any of the previous versions still exist in the application then accessing the deprecated keys on the merged version may throw JsonLiteException or return the wrong value. Therefore, it is important that all JsonLite objects of the previous versions must be upgraded to the merged version. This applies only to the previous versions with deprecated keys. To upgrade (or downgrade) objects, use the Pado management tool available via Pado Desktop.*

Deploying KeyType Jar Files

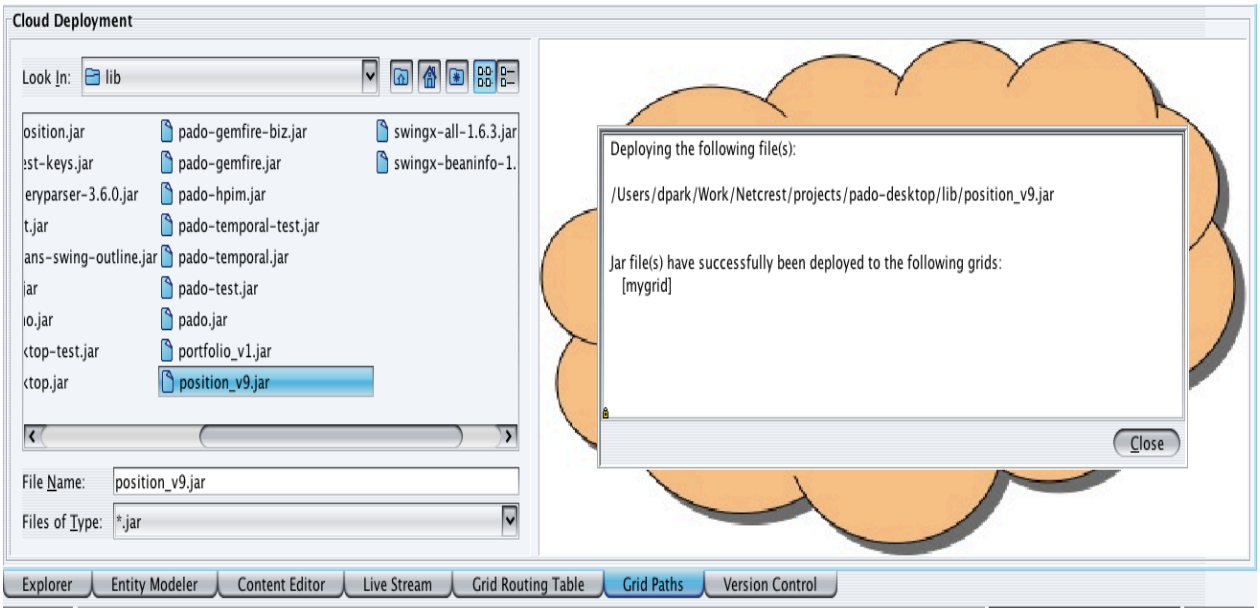
The `KeyType` classes should be jarred separately from your application classes so that they can be easily managed and deployed. You can segregate the `KeyType` classes in Eclipse, for example, by creating a separate source directory. The jar file that you create should contain only `KeyType` classes including the main and all versioned classes.

Once you have jar files ready, run the Pado's deploy command as follows:

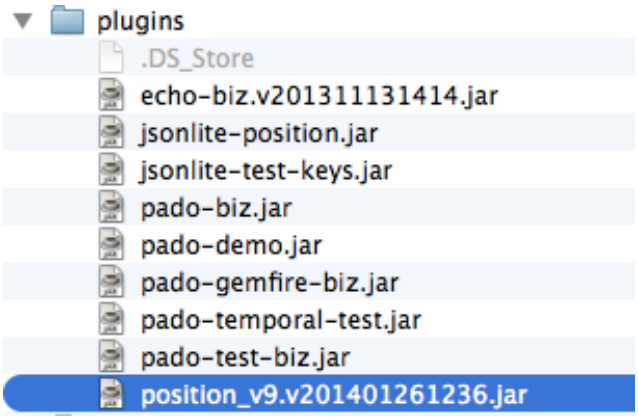
```
>cd bin_sh/tools

bin_sh/tools>./deploy -jar <comma separated jar paths>
```

The “deploy” command deploys the entire comma-separated jar files listed. The “deploy -dir" command deploys the entire jar files found in the specified directory.

Alternatively, you can use the Pado Desktop to deploy the jar files. Launch the Pado Desktop and login (See the Pado Quick Start document for instructions on launching Pado Desktop.) Once successfully logged on, select the "Grid Paths" tab to view the display as shown below. Select the jar file to deploy from the file browser component in the left pane and drag and drop it in anywhere in the cloud in the right pane. Upon successful deployment, you will see a message similar to shown in the diagram below.



Upon successful deployment, you will see a new jar file in the `$PADO_HOME/plugins` directory as shown below.



Note that the `position_v9.jar` has been renamed to `position_v9.v201401261236.jar`. The number indicates year 2014, month 01, date 26, hours 12, and minutes 36. Each time you deploy a jar file, its name will be extended to include the timestamp. When you restart Pado, it will load only the latest file with the newest timestamp.

Running Client Applications

To use `JsonLite`, non-Pado client applications only need to include the*pado-jsonlite.jar* file in the class path. This file is found in the `$PADO_HOME/lib` directory. For Pado client applications, JsonLite is included in the `$PADO_HOME/lib/pado.jar` file.

Note that it is possible for client applications to also use the deployment facility. Please contact Netcrest support for details.

