DOCUMENTATION

Liferay Portal

- Liferay Portal 6.1
- Liferay Portal 6.0
- Liferay Portal 5.2
- Liferay Portal 5.1

Liferay Social Office

- Liferay Social Office 2.0
- Liferay Social Office 1.5

Additional Resources

- Whitepapers
- Web Event Recordings

Documentation

Liferay provides a rich store of resources and knowledge to help our community better use and work with our technology.

User Guide Development Community Resources

Liferay Portal 6.1 - Development Guide

Previous - SOAP Web Services

Table of Contents »

Liferay APIs »

JSON Web Services

Next - Service Context

JSON Web Services

JSON Web Services provide convenient access to portal service methods by exposing them as JSON HTTP API. This makes services methods easily accessible using HTTP requests, not only from JavaScript within the portal, but also from any JSON-speaking client.

JSON Web Service functionality can be split into the following topics: registration, configuration, invocation and results. We'll cover each topic here.

Registering JSON Web Services

Liferay's developers use a tool called *Service Builder* to build services. All remote-enabled services (i.e. entities with `remote-service="true"` in `service.xml`) built with Service Builder are exposed as JSON Web Services. When Service Builder creates each `-Service.java` interface for a remote-enabled service, the `@JSONWebService` annotation is added on the class level of that interface. Therefore, *all* of the public methods of that interface become registered and available as JSON Web Services.

A `-Service` interface is a generated source file which is not to be modified by by the user directly. Sometimes, however, you need more control over which methods to expose and/or hide. To do so, just simply configure the `-ServiceImpl` class of the service. When service implementation class (`-ServiceImpl`) is annotated with the `@JSONWebService` annotation, the service interface is ignored and only the service implementation class is used for configuration. In other words, `@JSONWebService` annotations in service implementation **override** any JSON Web Service configuration in service interface.

And that's all! Upon start-up, Liferay Portal scans classes on the classpath for annotations. The scanning process is optimized so only portal and service JARs are scanned, as well as class raw bytecode content. Each class that uses the `@JSONWebService` annotation is loaded and further examined; its methods become exposed as JSON API. As explained previously, the `-ServiceImpl` configuration overrides the `-Service` interface configuration

during registration.

For example, let's look the `DLAppService`:

```
@JSONWebService
public interface DLAppService {
...
```

It contains the annotation found on portal startup. Notice the following lines in the console output when the debug log level is set:

```
10:55:06,595 DEBUG [JSONWebServiceConfigurator:121]
Configure JSON web service actions
10:55:06,938 DEBUG [JSONWebServiceConfigurator:136]
Configuring 820 actions in ... ms
```

At this point, scanning and registration is done and all service methods (those of `DLAppService` and of other services) are registered as JSON Web Services.

Registering Plugin JSON Web Services

Custom portlets can be registered and scanned for JSON web services, too. Services that use the `@JSONWebService` annotation become part of the JSON API. Since scanning of portlet services is not enabled by default, add the following servlet definition in your portlet's `web.xml`:

```
<web-app>
...
<filter>
<filter-name>Secure JSON Web Service Servlet Filter</filter-
name>
<filter-
class>com.liferay.portal.kernel.servlet.PortalClassLoaderFil
```

```
ter</filter-class>
<init-param>
<param-name>filter-class</param-name>
<param-
value>com.liferay.portal.servlet.filters.secure.SecureFilter
</param-value>
</init-param>
<init-param>
<param-name>basic_auth</param-name>
<param-value>true</param-value>
</init-param>
<init-param>
<param-name>portal_property_prefix</param-name>
<param-value>jsonws.servlet.</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>Secure JSON Web Service Servlet Filter</filter-
name>
<url-pattern>/api/jsonws/*</url-pattern>
</filter-mapping>

<servlet>
<servlet-name>JSON Web Service Servlet</servlet-name>
<servlet-
class>com.liferay.portal.kernel.servlet.PortalClassLoaderSer
vlet</servlet-class>
<init-param>
<param-name>servlet-class</param-name>
<param-
value>com.liferay.portal.jsonwebservice.JSONWebServiceServle
```

```
t</param-value>
</init-param>
<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>JSON Web Service Servlet</servlet-name>
<url-pattern>/api/jsonws/*</url-pattern>
</servlet-mapping>
...
</web-app>
```

This enables the servlet to scan and register your portlet's JSON Web Services.

Mapping and naming conventions

Mapped URLs of exposed service methods are formed using the following naming convention:

```
http://[server]:[port]/api/jsonws/[service-class-name]/[service-method-name]
```

where:

- **service-class-name** is the name generated from service class name, by removing the `Service` or `ServiceImpl` suffix and converting it to a lowercase name.
- **service-method-name** is generated from the service method name, by converting the camel-case method name to a lowercase separated-by-dash name.

For example, the following service method …

```
@JSONWebService
```

```
public interface UserService {
public com.liferay.portal.model.User getUserById(long
userId) {...}
```

… is mapped to the following URL:

```
http://localhost:8080/api/jsonws/user-service/get-user-by-id
```

Each service method is also bound to one HTTP method type. All methods having names starting with `get`, `is` or `has` are assumed to be read-only methods and are therefore mapped as GET HTTP methods, by default. All other methods are mapped as POST HTTP methods.

For plugins, you have two options for accessing their JSON Web Services.

Option 1 - Accessing the plugin service via the plugin context (e.g. your custom portlet's context):

```
http://[server]:[port]/[plugin-context]/api/jsonws/[service-
class-name]/[service-method-name]
```

However, this calls the plugin's service in a separate web application, that is not aware of the user's current session in the portal. As a result, accessing the service in this manner requires additional authentication.

Option 2 - Accessing the plugin service via the portal context:

```
http://[server]:[port]/[portal-context]/api/jsonws/[plugin-
context].[service-class-name]/[service-method-name]
```

Requests sent this way can conveniently leverage the user's authentication in his current portal session. Liferay's JavaScript API for services calls plugin services this way.

Listing available JSON Web Services

To overview a service and verify which service methods are registered and available, you can get a service listing in your browser by opening the base address:

```
http://localhost:8080/api/jsonws
```

The resulting page lists all registered and exposed service methods of the portal. You can see more details of each method by clicking the method name. For example, you can see the full signature of the service method, list of all its arguments, list exceptions that can be thrown, and even read its Javadoc! Moreover, you can even invoke the service method for testing purposes using simple form right from within your browser.

To list registered services on a plugin (e.g. a custom portlet), don't forget to use its context path:

```
http://localhost:8080/[plugin-context]/api/jsonws
```

This will list the JSON Web Service API for the portlet.

More on registration

As said, you can control registration by using annotations in your `-ServiceImpl` class. This overrides any configuration defined in the interface. Moreover, you can fine-tune which *methods* are visible/hidden using annotations at the method level.

Ignoring a method

To ignore a method from being exposed as a service, just annotate the method with:

```
@JSONWebService(mode = JSONWebServiceMode.IGNORE)
```

Any methods annotated like this do not become part of the JSON Web Service

API.

HTTP method name and URL

It is also possible to define custom HTTP method names and URL names, using a similar annotation at the method level.

```
@JSONWebService(value = "add-file-wow", method = "PUT")
public FileEntry addFileEntry(
```

In this example, the `DLApp` service method `addFileEntry` is mapped to URL method name `add-file-wow`. The complete URL is actually `http://localhost:8080/api/jsonws/dlapp/add-file-wow` and can be accessed using the PUT HTTP method.

If the URL name starts with a slash character (`/`), only the method name is used to form the service URL; the class name is ignored.

```
@JSONWebService("/add-something-very-specific")
public FileEntry addFileEntry(
```

Similarly, you can change the class name part of the URL, by setting the value in class-level annotation:

```
@JSONWebService("dla")
public class DLAppServiceImpl extends DLAppServiceBaseImpl {
```

This maps all the service methods of the class to URL class name `dla` instead of the `dlapp` default.

Manual registration mode

Up to now, it is expected most of the service methods are going to be exposed; that only specific methods are to be hidden (the *blacklist* approach). But sometimes you might need a different behavior: to explicitly specify only

those methods that are to be exposed (*whitelist* approach). This is possible, too, using so-called *manual mode* on class-level annotation. Then, it is up to you annotate only those methods which are to be exposed.

Then you can annotate only methods that have to be exposed.

```
@JSONWebService(mode = JSONWebServiceMode.MANUAL)
public class DLAppServiceImpl extends DLAppServiceBaseImpl {
...
@JSONWebService
public FileEntry addFileEntry(
```

Now only the `addFileEntry` method and any other method annotated with `@JSONWebService` are to be part of the JSON Web Service API; all other methods of this service are to be excluded from the API.

Next, let's take a look at portal configuration options that apply to JSON Web Services.

Portal Configuration of JSON Web Services

JSON Web Services are enabled on Liferay Portal by default but can be easily disabled by specifying the following portal property setting:

```
json.web.service.enabled=false
```

Strict HTTP methods

JSON Web Service services are, by default, mapped to either GET or POST HTTP methods. If a service method has a name that starts with `get`, `is` or `has`, the service is assumed to be read-only and is bound to the GET method; otherwise it is bound to POST.

By default, the portal does not check HTTP methods when invoking a service call; that is, the portal works in "non-strict http method" mode as services may

be invoked using any HTTP method. If you need the strict mode, you can set it with portal property:

```
jsonws.web.service.strict.http.method=true
```

When using strict mode, you must use the correct HTTP methods in calling service methods.

Disabling HTTP methods

When strict HTTP method mode is enabled, you can even filter web service access based on HTTP methods used by the services. For example, you can set the portal JSON Web Services to work in read-only mode by disabling HTTP methods other than GET. For example:

```
jsonws.web.service.invalid.http.methods=DELETE,POST,PUT
```

Now all requests that use HTTP methods from the list above are simply ignored.

Controlling public access

Each service method determines for itself whether it can be executed by unauthenticated users and whether a user has adequate permission for the chosen action. Most of portal's read-only methods are open to public access.

If you are concerned about the security, it is possible to additionally restrict the access to exposed JSON API for public access. For this reason, there is a property that specifies a comma delimited list of public methods that can be accessed by unauthenticated users.

```
jsonws.web.service.public.methods=*
```

Wildcards are supported, so, for example, you can simply set `get*,has*,is*` to only enable public access to read-only methods; additionally securing all other JSON methods. To disable access to all

exposed methods specify an empty value or to enable access to all exposed methods specify `*`.

Lastly, let's consider how to invoke JSON Web Services.

Invoking JSON Web Services

JSON Web Services can be invoked in several ways depending on how their parameters (i.e. method arguments) are passed in. But before we dive into different ways of passing parameters, it's important to understand how your invocation is matched to a method.

Matching service methods

It is important to understand how calls to service methods are matched, especially when a service method is overloaded.

The general rule is that besides the method name, you must provide **all** parameters for that service method. Even if some parameter is to be `null`, you must still provide it.

Note that how parameters are provided (as part of the URL line, as request parameters, etc.) is not important nor is the order of the parameters.

An exception to the rule of *all* parameters being required, is when using numeric *hints* to match methods. Let's look at using hints next.

Using hints

It is possible to add numeric hints that specify how many method arguments a service has. Hints are added as numbers separated by a dot in the method name. For example:

```
/foo/get-bar.2/param1/123/-param2
```

Here, the `.2` is a hint, so only service methods with 2 arguments will be

matched, others will be ignored for matching.

One important difference when a hint is specified, is now you do not have to specify all of the parameters. All missing arguments are treated as `null`. Therefore, the previous example may be called with …

```
/foo/get-bar.2/param1/123
```

… and `param2` will automatically be set to `null`.

Passing parameters as part of URL path

Parameters can be passed as part of the URL path. After the service URL, you can append methods parameters in name/value pairs. Parameter names must be formed from method argument names by converting them from camel-case to lowercase separated-by-dash names. Example:

```
http://localhost:8080/api/secure/jsonws/dlapp/get-file-entries/repository-id/10172/folder-id/0
```

Parameters may be given in **any** order; it's not necessary to follow the order in which the arguments specified in the method signatures.

When a method name is overloaded, the *best match* will be used: The method that contains the least number of undefined arguments is invoked.

Passing parameters as URL query

Parameters can be passed as request parameters, too. The difference is parameter names are specified as is (e.g. camel-case) and are set equal to their argument values:

```
http://localhost:8080/api/secure/jsonws/dlapp/get-file-entries?repositoryId=10172&folderId=0
```

As with passing parameters as part of a URL path, the parameter order is not important, the *best match* rule applies for overloaded methods, etc.

Mixed way of passing parameters

Parameters can be passed in a mixed way: some can be part of the URL path and some can be specified as request parameters.

Type conversion of the parameters

Parameter values are sent as strings using the HTTP protocol. Before a matching Java service method is invoked, each parameter value is converted from a string to its target Java type. We use a 3rd party open-source library to convert each object to its appropriate common type. Of course, it is possible to add or change the conversion for certain types; but we'll just cover how the conversions work by as-is.

Conversion for common types (`long`, `String`, `boolean`, etc.) is straightforward. All dates can be given in milliseconds. Locales, can be passed as locale names (e.g. `en` and `en_US`). To pass in an array of numbers, send a string of comma-separated numbers (e.g. string `4,8,15,16,23,42` can be converted to `long[]` type). You get the picture!

Arguments can be of type `List` or `Map`, too! To pass a `List` argument, send a JSON array. To pass a `Map` argument, send a JSON object. The conversion then is done in two steps:

*Step 1 - JSON deserialization - * JSON arrays are converted into `List<String>` and JSON objects are converted to `Map<String, String>`. Note, due to security reasons, it is forbidden to instantiate any type within JSON deserialization.

*Step 2 - Generification - * Each `String` element of the `List` and `Map` is converted to its target type (the argument's Java generics type specified in the method signature). Note, this step is only done if the Java argument type uses generics.

For example, let's consider the conversion of string array `[en,fr]` as JSON web service parameters for a `List<Locale>` Java method argument type:

*Step 1 - JSON deserialization - * The JSON array is deserialized to a `List<String>` containing strings `en` and `fr`.

*Step 2 - Generification - * Each string is converted to the `Locale` (the generics type), resulting in the `List<Locale>` Java argument type.

Sending NULL values

To pass a `null` value for an argument, simply prefix the parameter name with a dash `-`. For example:

```
.../dlsync/get-d-l-sync-update/company-id/10151/repository-id/10195/-last-access-date
```

Here the `last-access-date` parameter is interpreted as `null`.

Null parameters, therefore, do not have specified values. Of course, null parameters do not have to be the last in the URL, as in this example. When a null parameter is passed as a request parameter, its value is ignored and `null` is used instead:

```
<input type="hidden" name="-last-access-date" value=""/>
```

When using JSON RPC (see below), null values may be sent explicitly, even without a prefix. For example:

```
"last-access-date" : null
```

Parameters encoding

Although often forgotten, there is a difference between URL encoding and

query (i.e. request parameters) encoding. An illustrative example of this is the difference in how the space character is encoded. When the space character is part of the URL path, it is encoded as `%20`; when it is part of the query it is encoded as plus sign (`+`).

Furthermore, all of these rules for encoding apply to international (non-ascii) characters, as well. Since Liferay Portal works in UTF-8 mode, parameter values must be encoded as UTF-8 values. However, the portal itself is not responsible for decoding request URLs and request parameter values to UTF-8. This task is done by the web-server layer (Tomcat, Apache, etc.). When accessing services through JSON-RPC, encoding parameters to UTF-8 is not enough – we also need to send the encoding type in a Content-Type header (e.g. `Content-Type : "text/plain; charset=utf-8"`).

For example, let's pass the value "Супер" ("Super" in Cyrillic) to some JSON Web Service method. This name first has to be converted to UTF-8 (resulting in array of 10 bytes) and then encoded for URLs or request parameters. The resulting value is the string `%D0%A1%D1%83%D0%BF%D0%B5%D1%80` that can be passed to our service method. When received, this value is first going to be translated to an array of 10 bytes (URL decoded) and then converted to a UTF-8 string of the 5 original characters.

Sending files as arguments

Files can be uploaded using multipart forms and requests. Example:

```
<form
action="http://localhost:8080/api/secure/jsonws/dlapp/add-
file-entry" method="POST" enctype="multipart/form-data">
<input type="hidden" name="repositoryId" value="10172"/>
<input type="hidden" name="folderId" value="0"/>
<input type="hidden" name="title" value="test.jpg"/>
<input type="hidden" name="description" value="File upload
example"/>
<input type="hidden" name="changeLog" value="v1"/>
```

```
<input type="file" name="file"/>
<input type="submit" value="addFileEntry(file)"/>
</form>
```

As you see, it's a common upload form that invokes the `addFileEntry` method of the `DLAppService` class.

JSON RPC

JSON Web Service may be invoked using JSON RPC. A good part of JSON RPC 2.0 specification is supported in Liferay JSON Web Services. One limitation is parameters may be passed only as *named* parameters; positional parameters are not supported, as there are too many overloaded methods for convenient use of positional parameters.

Here is an example of invoking a JSON web service using JSON RPC:

```
POST http://localhost:8080/api/secure/jsonws/dlapp
{
"method":"get-folders",
"params":{"repositoryId":10172, "parentFolderId":0},
"id":123,
"jsonrpc":"2.0"
}
```

Default parameters

When accessing *secure* JSON web services (user has to be authenticated), some parameters are made available to the web services by default. They need not to be specified explicitly, unless you want to change their values to something other than their defaults.

Default parameters are:

- `userId` - id of authenticated user
- `user` - full User object
- `companyId` - users company
- `serviceContext` - empty service context object

Object parameters

Most services accept simple parameters: numbers, strings etc. However, sometimes you need to provide an object (a non-simple type) as a service parameter.

Similar to specifying null parameters by using the `-` prefix, to create an instance of an object parameter, just prefix the parameter name with a plus sign, `+`, without any parameter value at all. For example:

```
/jsonws/foo/get-bar/zap-id/10172/start/0/end/1/+foo
```

or as a request parameter (note, the `+` sign must be encoded!):

```
/jsonws/foo/get-bar?zapId=10172&start=0&end=1&%2Bfoo
```

or

```
<input type="hidden" name="+foo" value=""/>
```

If a parameter is an abstract class or an interface, it can't be instantiated as such. Instead, a concrete implementation class must be specified to create the argument value. This can be done by specifying the `+` prefix before the parameter name followed by specifying the concrete implementation class. For example:

```
/jsonws/foo/get-bar/zap-
id/10172/start/0/end/1/+foo:com.liferay.impl.FooBean
```

or

```
<input type="hidden" name="+foo:com.liferay.impl.FooBean"
value=""/>
```

The examples above specify that a `com.liferay.impl.FooBean` object, presumed to implement the class of the parameter named `foo`, is to be created.

A concrete implementation can be set as a value, too! For example:

```
<input type="hidden" name="+foo"
value="com.liferay.impl.FooBean"/>
```

or in JSON RPC:

```
"+foo" : "com.liferay.impl.FooBean"
```

All these examples specify a concrete implementation for `foo` service method parameter.

Inner Parameters

In many cases, you'll need to populate objects that are passed as parameters. A good example is a default parameter `serviceContext` of type `ServiceContext` (see the *Service Context* section in this chapter). Sometimes, you need to set some of the inner properties (i.e. fields) of the `ServiceContext`, such as: `addGroupPermissions`, `scopeGroupId`, etc., to make an appropriate call to a JSONWS.

To pass inner parameters, just specify them using a 'dot' notation. That is, specify the name of the parameter, followed by a dot `.`, followed by the name of the inner parameter. For example, with regards to the `ServiceContext` inner parameters you can provide: `serviceContext.addGroupPermissions`, `serviceContext.scopeGroupId` parameters, together with other parameters of your web service call. They will be recognized as inner

parameters (with a dot in the name) and their values will be *injected* into existing parameters, before the API service method is executed.

Inner parameters are not counted as *regular* parameters for matching methods and are ignored during matching.

Tip: Use inner parameters with object parameters to set inner content of created parameter instances!

Returned values

No matter how a JSON web service is invoked, it returns a JSON string that represents the service method result. Any returned objects are *loosely* serialized to a JSON string and returned to the caller.

Let's take a look a some returned values from calls to services. In fact, let's create a UserGroup as we did in our previous SOAP web service client examples. To make it easy, we'll use the test form provided with the JSON web service in our browser.

1. Open your browser to the JSON web service method that adds a UserGroup:

   ```
   http://127.0.0.1:8080/api/jsonws?
   signature=/usergroup/add-user-group-2-name-description
   ```

   or navigate to it by starting at `http://127.0.0.1:8080/api/jsonws`,

scrolling down to the section for *UserGroup* and clicking *add-user-group*.

2. Fill in the *name* field to “MyUserGroup3” and the *description* to some arbitrary value string like “Created using JSON WS”.

3. Click *Invoke* to get a result similar to the following:

```
{"addedByLDAPImport":false,"companyId":10154,"description":"Created using JSON
WS","name":"MyUserGroup33","parentUserGroupId":0,"userGroupId":13162}
```

Notice the JSON string returned represents the `UserGroup` object you just created. The object has been serialized into a JSON string. As a starting point for understanding JSON strings, go to json.org.

To find out how to serialize Java objects, maps and lists, check out article JSON Serialization by Igor Spasić.

Common JSON WebService errors

While working with JSON Web Services, you may encounter some of the common errors described in the following subsections.

Missing value for parameter

This error means you didn’t pass a parameter value along with the parameter name in your the URL path. The parameter value must follow the parameter name in the URL path.

Here is an example, the URL path

```
/api/jsonws/user/get-user-by-id/userId
```

specifies the parameter named `userId`, but it does not specify the parameter's value. To resolve this error, simply provide the parameter value after the parameter name:

```
/api/jsonws/user/get-user-by-id/userId/173
```

No JSON web service action associated

This is error means no service method could be matched with the provided data (method name and argument names). This can be due to various reasons: arguments may be misspelled, the method name may be formatted incorrectly, etc. Since JSON web services reflect the underlying Java API, any changes in the respective Java API will automatically be propagated to the JSON web services. For example, if a new argument is added to a method or an existing argument is removed from a method, the parameter data must match that of the new method signature.

Unmatched argument type

This error appears when you try to instantiate a method argument using an incompatible argument type.

JSON Web Services Invoker

Using JSON Web Services is easy, you send a request that defines a service method and parameters and you receive the result as JSON object. But you may need to use JSON Web Services more pragmatically.

Consider the following example: You are working with two related objects, a `User` and its corresponding `Contact`. With simple JSON Web Service calls, you first call some user service to get the user object and then you call the contact service using the contact ID from the user object. So you end up sending two HTTP requests to get two JSON objects that are not even bound together; there is no contact information in the user object (i.e. no `user.contact`). Obviously, this approach has some impact on performance (sending two HTTP calls) and on usability (manually managing the

relationship between two objects). Wouldn't it be nice if you had an easy-to-use tool to address these problems? Well, you do – the *JSON Web Service Invoker*.

Liferay's JSON Web Service Invoker helps you optimize your use of JSON Web Services. In the following sections, we'll show you how.

A simple Invoker call

The Invoker is accessible on the fixed address:

```
http://[address]:[port]/api/jsonws/invoke
```

It only accepts one request parameter: `cmd` – the Invoker's command. If the command request parameter is missing, the request body is used as the command. So, basically, you can specify the command by either using the request parameter `cmd` or the request body.

The Invoker command is a plain JSON map that describes how JSON Web Services are to be called and how the results are to be managed. Here is an example of how to call a simple service using the Invoker:

```
{
"/user/get-user-by-id": {
"userId": 123,
"param1": null
}
}
```

As you can see, the service call is defined as a JSON map. The key specifies the service URL (i.e. the service method to be invoked) and the key's value specifies a map of service parameter names (i.e. `userId` and `param1`) and their values. In the example above, the retrieved user is returned as a JSON object. Moreover, since the command is a JSON string, null values can be specified explicitly using the `null` keyword. However, if you so choose, you

can still use the less natural convention for specifying a null parameter that requires a dash before the parameter name and an explicit empty value (e.g. `"-param1": ''`).

Note, the example Invoker call is identical to the following standard JSON Web Service call:

```
/user/get-user-by-id?userId=123&-param1
```

Before we dive into more features, let's learn how to use variables with the Invoker.

Invoker variables

Variables are used to reference objects returned from service calls. Variable names must start with a `$` (dollar sign) prefix. In our previous example, the service call returned a user object that can be assigned to a variable:

```
{
"$user = /user/get-user-by-id": {
"userId": 123,
}
}
```

Here, the variable `$user` holds the returned user object. You can reference the user's contact ID using the syntax `$user.contactId`.

Nested service calls

With nested service calls, you can magically bind information from related objects together in a JSON object. This feature allows you to not only call other services within the same HTTP request, but also nest returned objects in a convenient way. See it in action:

```
{
```

```
"$user = /user/get-user-by-id": {
"userId": 123,
"$contact = /contact/get-contact-by-id": {
"@contactId" : "$user.contactId"
}
}
}
```

This command defines two service calls in which the contact object returned from the second service call is nested in (i.e. injected into) the user object, as a property named `contact`. Finally, we can bind the user and its contact information together!

Let's analyze this command example to consider what the JSON Web Service Invoker does in the background within this single HTTP request:

- Calls the Java service mapped to `/user/get-user-by-id` passing in a value for the `userId` parameter
- Assigns the resulting user object to variable `$user`
- Proceeds with invoking nested calls
- Calls the Java service mapped to `/contact/get-contact-by-id` using `contactId` parameter, with `$user.contactId` value from `$user` object
- Assigns the resulting contact object to variable `$contact`
- Injects the contact object referenced by `$contact` into the user object's property named `contact`

One remark: you need to *flag* parameters that take values from existing variables. Flagging is done using the `@` prefix before the parameter name.

Filtering results

Many of Liferay Portal's model objects are rich with properties. But, you may only need a handful of an object's properties for your business logic. By reducing the number of properties returned in an object, you can minimize the

network bandwith used by your web service invocation. Good news! With the JSON Web Service Invoker you can define a *white-list* of properties to include only specific properties in the object returned from your web service call. It's simple:

```
{
"$user[firstName,emailAddress] = /user/get-user-by-id": {
"userId": 123,
"$contact = /contact/get-contact-by-id": {
"@contactId" : "$user.contactId"
}
}
}
```

In this example, the returned user object has only the `firstName` and the `emailAddress` properties (and, of course, the `contact` property). You specify *white-list* properties in square brackets (`[...]`) immediately following the name of your variable.

Batching calls

As mentioned previously, nesting service calls allows you to invoke multiple services within a single HTTP request. Using a single request for multiple service calls is helpful for gathering related information from the service call results. But you can also use a single request to invoke unrelated service calls. The Invoker command allows you to *batch* service calls together to improve performance. Again, it's simple, just pass a JSON array of commands:

```
[
{/* first command */},
{/* second command */}
]
```

The result is a JSON array populated with results from each of the commands. The commands are collectively invoked in a single HTTP request, one after another.

Well, you've just added some powerful tools to your toolbox by learning how to leverage JSON Web Services in Liferay. Good job!

Next, let's consider the `ServiceContext` class used by so many Liferay services and how it can be helpful to use in your services.

View »

Previous - SOAP Web Services

Table of Contents »

Liferay APIs »

JSON Web Services

Next - Service Context