# Mate: Mock services

Mate Mock services lets you replace any RemoteObject or WebService with a mock service that will behave as if the actual service was called, returning a result that you create based on the service arguments.

To mock a RemoteObject, use the MockRemoteObject tag and to mock a WebService, use the MockWebService tag. Both have the same attributes and inner tags.

## Attributes

**mockGenerator**
Class to instantiate that will generate the mock result.
This attribute needs to be supplied here or in the individual MockMethod tags.

**delay**
Number of seconds to take to return the result or fault after making the call. This helps making the illusion that the service is taking time to respond. Default is 0 (no delay).

## Inner tags

## MockMethod

## Attributes

**name**
name of the service method that we are mocking. If the RemoteObject would normally have a method called "getData", then the name attribute should be "getData".

**mockGeneratorMethod**

name of the method on the mock generator class that will handle the service call and return a result. This method should accept the same parameters as the actual service method, plus an optional last parameter to receive any loaded data (when dataUrl is supplied). If not supplied, then the mock service will attempt to call a method with the same name as the "name" attribute.

**delay**
Number of seconds to take to return the result or fault after making the call. Same as the MockRemoteObject and MockWebService's delay attribute. It will overwrite any delay set in the

container tag.

**mockGenerator**
Class to instantiate that will generate the mock result.
Same as the mockGenerator attribute of the MockRemoteObject and MockWebService. When supplied, this class will be used instead of the one set in the container tag.

**dataUrl**

If you wish to load some data before returning the result, you can supply this attribute and the mock service will load this data before calling the method on your mockGenerator class. The data loaded will be passed as the last argument on the method call.

**async**
true/false
If true, then the mock method will have to dispatch a ResultEvent or a FaultEvent when it is ready. If false, then the mock method needs to supply the desired result as the return value. If you don't want return a result, you can throw an error. That error will be transformed into a fault.

Example

Suppose your RemoteObject service the following methods:
getEmployeesByDepartment(departmentId) that returns an array of Employee objects
saveEmployee(employee) that returns the updated employee and
deleteEmployee(employee) that returns void.

To mock this service, you have several options. You will always need a separate class that will decide what data to return and how to actually mock the service.

**Mock generator matching methods option**

The first option is the simplest. You will create a class that will mock the services. As an example, we'll call it "EmployeeMockHelper". This class will need to have functions matching the remote service functions, in name, number and type of parameters they receive and return type:

```
public class EmployeeMockHelper {
```

```
        public function getEmployeesByDepartment(id:int):Array {
            //create some employee mock objects
        }


        public function saveEmployee(employee:Employee):Employee{
            //return an employee
        }


        public function deleteEmployee(employee:Employee):void{
        }
}
```

Within those functions, you would create mock data, and return what a real service would return, such as an array of employee objects or an employee.

Then to use this mock helper class, you will replace your real remote object tag:

```
<mx:RemoteObject id="service" ... />
```

for a MockRemoteObject, and then use normally. You will need to supply the mockGenerator attribute with the mock helper class you created:

```
<MockRemoteObject id="service"
mockGenerator="{EmployeeMockHelper}" />
```

As long as your functions return data of the same type as your remote services, all your existing code will work. If you wish to indicate a fault instead of a result, you can throw an error:

```
throw new Error('service not available');
```

Any error will be converted to a fault.

**Mock generator not matching methods**

If your mock helper class has names that do not match the service names, then you can specify the method name mapping by using the MockMethod inner tag. By using the MockMethod tag, you have finer control on what methods should be called, and whether a specific method should have a longer or shorter delay.

If your mock helper class had a method called "save", instead of "saveEmployee", you would map

them as follows:

```
<MockRemoteObject id="service"
mockGenerator="{EmployeeMockHelper}">
     <MockMethod name="saveEmployee" mockGeneratorMethod="save"
/>
</MockRemoteObject>
```

The save method will still need to accept the same parameters as the service saveEmployee function.

**Different Mock generator for a particular method**

The mockGenerator attribute for the MockRemoteObject and MockWebservice tags applies to all methods in the service, but if you need to have a different mock class for a particular method, you can specify it in the MockMethod tag:

```
<MockRemoteObject id="service"
mockGenerator="{EmployeeMockHelper}">
     <MockMethod name="saveEmployee"
mockGenerator="{OtherMockHelper}" mockGeneratorMethod="save" />
</MockRemoteObject>
```

**Loading data for the mock helper**

If your mock helper class needs data to be loaded via an http request (ie: an XML file containing mock records), you can specify the url with the dataUrl attribute, and the mock service will load that file and after the data is available, it will call the method on the mock generator. This data will be passed as the last parameter of the function call, in addition to the parameters required by the service call.

Suppose you have the list of employees in an XML file you wish to load before returning the employees in the getEmployeesByDepartment(departmentId) method. You will need a method in your mock helper class that either matches the name or that you map with the MockMethod tag. This method needs to accept the department id, and also a second parameter to receive the data that was loaded:

```
public function getEmployeesByDepartment(id:int,
data:String):Array {
     //parse data xml, and create some employee mock objects
```

```
}
```

**Letting your mock helper class dispatch results and faults**

If you wish to load data or perform any asynchronous task before returning a result, such as call an actual service, you need to specify true in the "async" attribute of the MockMethod tag. If you do so, your mock helper method needs to dispatch a ResultEvent or FaultEvent. Failing to do so will maintain the service waiting for a result or fault and, if you are using Mate, your resultHandlers and faultHandlers will never execute.

When your method is ready to dispatch a result, for example after loading some data, you would write:

```
this.dispatchEvent(ResultEvent.createEvent(myEmployeeArray));
```