

Getting started with Ctx framework

By Andrii Olefirenko, July 2009

About the framework

Ctx is an dependency injection framework for ActionScript3/Flex projects. Ctx support property injections (autowired by id and by type) and has many functions to help with asynchronous service methods, forms processing, data caching, testing etc.

Hello World! project

This tutorial describes how to create simple Hello world project. "Hello" service could be anything that extends AbstractService (HTTP, Webservice, AMF service). Here we'll be using MockService which imitates server functionality. MockService and MockOperation framework classes mostly used for testing or for "offline" development.

Alternatively you could download the Hello World! Project from the Ctx framework site.

What you need

1. Flex Builder 3 or Flash Builder (or any other IDE for ActionScript3/Flex)
2. Ctx.swc (added to Project Build Path)

Steps to create "Hello world" (using Flex Builder)

Steps

Create new Flex application project "CtxExamples"

Create new ActionScript class HelloMockRemoteObject that extends MockRemoteObject:

```
package examples.helloWorld
{
    import framework.MockRemoteObject;

    public class HelloMockRemoteObject extends MockRemoteObject
    {
        public function HelloMockRemoteObject()
        {
            addMockOperation("sayHello", sayHello, 400);
        }

        private function sayHello(name:String):String
        {
            return "Hello, " + name;
        }
    }
}
```

This will be our asynchronous service method (where "400" is delay in milliseconds for our "backend")

Create HelloModelBean. This will be our model class to store all data.

```
package examples.helloWorld
{
    public class HelloModelBean
    {
        [Bindable]
        public var helloText:String = "Hello";

        public function HelloModelBean()
        {
        }
    }
}
```

Just make sure that properties are bindable

Create class HelloService:

```
package examples.helloWorld
{
    import framework.handleData;

    import mx.rpc.AbstractService;

    public class HelloService
    {
        public var helloRemoteObject:AbstractService;

        public var $hmb:HelloModelBean;

        public function HelloService()
        {
        }

        public function sayHello(name:String):void
        {
            handleData(helloRemoteObject.sayHello(name), $hmb, "helloText");
        }
    }
}
```

This is more interesting class. `helloRemoteObject` property will be autowired by framework. The name of the property should be the same as bean ID. "Bean" is simply any AS3 class registered with framework. We'll see how to register classes as beans later. Bean ID is unique identifier for a bean. So `helloRemoteObject` property is autowired by ID. Basically the framework tries to autowire all public properties of the bean.

`$hmb` property name starts with "\$". It means that it will be autowired by property type rather than by property id. This is more convenient when you have only one bean of specific concrete type registered with framework.

`handleData()` is framework function that is applied to any asynchronous function (they return `AsyncToken` object) and listens for `ResultEvent`. Then it stores the result in specified property of some object. In our case it's `$hmb.helloText` property.

Some other functions:

```
public function handle(asyncToken:AsyncToken, resultHandler:Function,  
faultHandler:Function = null):AsyncToken
```

Used to handle asynchronous call (of any type:Http, Webservice, RemoteObject) using closures. You don't have to deal with event listeners (adding and removing). And what is more interesting you could apply many "handle" functions to the same asynchronous call.

```
public function branch(...args):AsyncToken
```

This function is used when you need to call more than one asynchronous method and you want to create token for all these calls (and don't really care about the order).

For example:

```
    handle(branch(serviceObject.loadCustomers(),  
    serviceObject.loadSuppliers()), onAllDataLoaded);
```

In this example, onAllDataLoaded function will be executed when

serviceObject.loadCustomers() and serviceObject.loadSuppliers() have returned their results.

```
public function chain(...args):AsyncToken
```

This function will execute asynchronous calls one by one (next call is executed when the previous has returned result successfully, so the calls are deferred)

For example:

```
handle(chain(userService.login, [username, password],  
userService.getCurrentUserProfile, []), onUserProfileLoaded)
```

In this case userService.login(username, password) will be executed first, then, after successful result, userService.getCurrentUserProfile() gets called.

onUserProfileLoaded is called when the last function has returned successful result.

Create class HelloCtx. This will be our context configuration and access class.

```
package examples.helloWorld  
{  
    import framework.Ctx;  
  
    public dynamic class HelloCtx extends Ctx  
    {  
        public function HelloCtx()  
        {  
            super("defaultContext");  
        }  
  
        override protected function registerBeans():void  
        {  
            register>HelloModelBean, "helloModelBean");  
            register>HelloService, "helloService");  
            register>HelloMockRemoteObject, "helloRemoteObject");  
        }  
    }  
}
```

Note that this class extends Ctx class. This is optional but useful if you want to use this class to access the context. Context is the place where all you beans are stores. You could have many contexts. Override

registerBeans function to define all beans (this function will be called only once for all instances). You could register beans later and in different places if needed.

```
public function register(clazzOrInstance:Object, id:String = null):*
```

This function is used to register classes so they will be managed by framework.

Some examples:

```
var c:Ctx = new Ctx;
c.register>HelloModelBean, "helloModelBean");
c.register("Simple text value", "textBean");
trace(c.textBean); //outputs "Simple text value"
c.deleteBean("helloModelBean");
delete c.textBean; // the same as deleteBean

c.newBean =>HelloService; // another way to register bean
c.anotherNewBean = new>HelloMockRemoteObject;
```

You can create as many Ctx instances as needed. They always will be referencing the same beans.

Optional. You can create getters for all you beans so FlexBuilder will be able to provide autocompletion.

```
public function get helloModelBean():HelloModelBean
{
    return getBean("helloModelBean");
}
```

And finally, application mxml file:

```
<?xml version = "1.0" encoding = "utf-8"?>
<mx:Application xmlns:mx = "http://www.adobe.com/2006/mxml"
    xmlns:helloWorld = "examples.helloWorld.*">
    <helloWorld>HelloCtx id = "c"/>
    <mx:Label text = "Enter your name"/>
    <mx:TextInput id = "yourName"/>
    <mx:Button label = "Say Hello"
        click = "{c.helloService.sayHello(yourName.text) }"/>
    <mx:Label text = "Result: {c.helloModelBean.helloText}"/>
</mx:Application>
```

<helloWorld>HelloCtx id = "c"/> you just create an instance of your config&access class, and access beans as properties of this instance. You can bind to properties or call beans' methods.

Conclusion

This is just a fraction of all functionality provided by Ctx framework. What is important that there is only one mandatory class – Ctx, others are optional – you can use them or not. There are also many utility functions to help you with asynchronous calls, form processing, data caching and so on.