## Hyperty Development

This document provides guidelines for developers of Hyperties. It is recommended that you have already read:

* [An Overview of the Hyperty Concept](hyperty.md)
* [An Overview of the Hyperty Messaging Framework](hyperty-messaging-framework.md)
* [An overview on how Hyperties cooperate with each other through a Data Synchronisation model called Reporter - Observer](p2p-data-sync.md)
* [An overview on the Hyperty Security and Trust Model](hyperty-trust.md)

### Hyperty Concept

The Hyperty Concept is introduced [here](hyperty.md) as a secure user associated [microservice](http://martinfowler.com/articles/microservices.html), which can be deployed either on a web runtime environment, on an end-user device or on a networked server. The main characteristics of a Hyperty include:

* *Modular*: A Hyperty should be a self contained module providing reusable service logic implementions in form of a script (e.g. a JavaScript file)
* *User association* : A Hyperty instance is associated to a “User” (e.g. Human beings, physical objects, physical spaces, organizations) through an Identity, even if this User can be anonymous in some cases.
* *Data Synchronization Communication*: Hyperties interact with each other through data synch objects by using the Reporter – Observer communication pattern.
* *Protocol Agnostic*: Through the protocol-on-the-fly concept, Hyperties are network protocol agnostic. In other words, the data synchronization communication between Hyperties is not dependent on a specific network protocol. Communication is accomplished via a common data schema that describes the data synch objects used.
* *GUI independent*: Hyperty should not provide Graphical User Interfaces. *to be clarified*
* *APIs*: A Hyperty can expose Javascript APIs within the runtime environment that can to be used by web applications

While designing and specifying service logics, it should be noted that Hyperties are not suitable for all use cases. In some case, making use of a simple resusable JavaScript file as library may suffice. The next section explaines the criteria under which the decision to use a Hyperty or not could apply.

### Criteria to use the Hyperty Concept

These are guidelines to help developers decide if they should provide specific service logic as Hyperty or via a simple JavaScript library. Consider these as guidelines and not misinstruction. Before you embark on a new feature development, ask yourself the following questions:\* Is the feature delivery directly associated with a user?\* Does the feature delivery involve communication between users?\* Is the feature modular and reusbale on different applications?\* Can the feature be delivered and developed by different stakeholders (i.e domain specific implementation)?

If the answers to the above questions are "YES" then most likely, you should go for the Hyperty Concept. The reTHINK Service Framework is what you want to look at next. The Service Framework provides APIs for developers to facilitate the development of Hyperties.

### Getting Started with the Service Framework

So you have decided for the Hyperty Concept and now ask yourself where to start. This section describes the basic steps any developer needs to undertake to include the Service Framework into their projects. There are two simple steps to get you started.

1. Install the Service Framework

npm install git+git@github.com:reTHINK-project/dev-service-framework.git#develop --save  
jspm install service-framework=github:reTHINK-project/dev-service-framework.git@develop

1. Import Module(s)

import {Syncher} from 'service-framework';

or if you need more than one dependency:

import {Syncher, MessageFactory, AddressFactory} from 'service-framework';

The next section explains the availble modules and APIs they expose.

### APIs

Here we describe useful functionalities that are exposed by the Service Framework Module, which developers can use in development process.

#### Syncher API

The Syncher is a singleton class per Hyperty/URL and it is the owner of all created DataObjects. The main class for the package. Should only be available one per Hyperty/URL. It's the owner of all kind of DataObjects.

new Syncher(hypertyURL, bus.MiniBus, configuration)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| hypertyURL | URL.HypertyURL | A URL allocated by the runtime that uniquely identifies the Hyperty |
| bus.MiniBus | MiniBus | An instance of the MiniBus provided in the sandbox. When an object (Reporter or Observed) is created, the SyncherManager will add a listener in the MiniBus to receive/send Messages of that object. |
| configuration | Config | Configuration data containing the runtimeURL. |

##### Methods

The create method request a DataObjectReporter creation. The URL will be be requested by the allocation mechanism..

create(schema, List, initialData)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| schema | Schema | The Schema of the object |
| List | Array | of hyperties to send the create |
| initialData | JSON | Object initial data |

* Returns: Return Promise to a new Reporter. The reporter can be accepted or rejected by the PEP Type Promise.

The subscribe method can be used to request subscription to an existent object.

subscribe(objURL)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| objURL | ObjectURL | Address of the existent object |

*Returns:* Return Promise to a new Observer of Type Promise.

#### Minibus API

The MiniBus API is a minimal interface to send and receive messages. It can be reused in many type of components. Components that need a message system should receive this class as a dependency or extend it. Classes extending this interface have to implement the following private methods: \_onPostMessage and \_registerExternalListener which are described below.

The \_onPostMessage method is a private class and used by the classes extending the Minibus class to process messages from the public "postMessage" without a registered listener. It can be used to send the message to an external interface, like a WebWorker or an IFrame.

onPostMessage(msg)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| msg | Message.Message | posted Message |

The \_registerExternalListener() method is not publicly available. It can be used by the class extension implementation to process all messages that enter the MiniBus from an external interface, like a WebWorker or IFrame. This method is called one time in the constructor to register external listeners. The implementation will probably call the \_onMessage method to publish in the local listeners.

*NOTE:* DO NOT call "postMessage", there is a danger that the message enters in a cycle!registerExternalListener()

#### Hyperty Discovery API

Hyperty Discovery interface provides the functionality to query hyperties instances registered in the domain registry of a given user

new HypertyDiscovery(domainURL, msgBus)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| domainURL | URL.RuntimeURL | A URL allocated by the runtime that uniquely identifies the Hyperty |
| msgBus.MiniBus | MiniBus | An instance of the MiniBus used to post messages to the Message Bus |

##### Methods

The discoverHypertyPerUser function is used to query hyperties instances registered in Domain registry for a given user.

discoverHypertyPerUser(userIdentifier)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| userIdentifier | Identity.Identity | The user's unique identifier |

* Returns:\* Return Promise

### Examples

#### Syncher Example

Here is an example on how a Hyperty can instantiate and use the syncher.

import {Syncher, MessageFactory} from '../src/service-framework';  
  
class MyAwesomeHyperty{  
  
 constructor(hypertyURL, bus, configuration)  
 {  
 let \_this = this;  
 \_this.bus = bus;  
 \_this.configuration = configuration;  
 \_this.hypertyURL = hypertyURL;  
 // Syncher Object  
 let syncher = new Syncher(hypertyURL, bus, configuration);  
 \_this.syncher = syncher;  
  
 //MessageFactory Object  
 let messageFactory = new MessageFactory("false", '{}');  
 \_this.messageFactory = messageFactory;  
  
 \_this.syncher.onNotification(function(event) {  
 console.log('My Awesome Hyperty just recieved a notification: ', event);  
 \_this.hypertyConnector.\_onNotification(event, hypertyURL);  
 });  
 \_this.hypertyConnector = new HypertyConnector(syncher);  
 \_this.hypertyConnector.name = 'My Awesome Hyperty';  
 }  
}

#### MiniBus API Example

We shall now provide more functionality to our MyAwesomeHyperty example above. The above class already has an instance of the MiniBus object which was provided in the constructor parameter. The example below shows how to use this instance to send a Message on the Message Bus.

sendMessage() {  
 let \_this = this;  
 let message = messageFactory.createCreateMessageRequest( \_this.hypertyURL,  
 'hyperty-runtime://sp1/AnotherHyperty'  
 "Hello from My Awesome Hyperty");  
 \_this.bus.postMessage(message);  
 }