## 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.
* *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 reusable JavaScript file as library may suffice. The next section explains 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 instructions. Before you embark on a new feature development, ask yourself the following questions:

* Is the feature delivery directly associated with a "User" (e.g. Human beings, physical objects, physical spaces, organizations)?
* Does the feature delivery involve communication between "Users"?
* Can the feature be delivered in cross-domain scenarios (i.e interoperability with other domains)?

If at least you have one "YES" answers to the above questions then most likely, you should go for the Hyperty Concept. Some examples are:

* Video Chat between human beings;
* a Human-being or a Data Analysis service collecting data from sensors in a Room.
* ..

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

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 develop Hyperties. The source code of the Hello World Hyperty used as example, can be found [here](https://github.com/reTHINK-project/dev-hyperty/tree/master/src/hello-world)

1. Clone the [Hyperty Toolkit Repo](https://github.com/reTHINK-project/dev-hyperty-toolkit/blob/master/README.md), the [Hyperty Repo](https://github.com/reTHINK-project/dev-hyperty) and Install the toolkit:

npm install

Create a new folder for your Hyperty at src folder and work there.

1. Select or specify the descriptor of the Hyperty Data Objects in json-schema. Let's assume we define a simple Hello Hyperty object:

{  
 "$schema": "http://json-schema.org/draft-04/schema#",  
 "id": "HelloObject",  
 "type": "object",  
 "required": ["scheme","hello"],  
 "additionalProperties": false,  
 "properties": {  
 "scheme": {  
 "constant": "hello"  
 },  
 "hello": {  
 "type": "string"  
 }  
 }  
}

1. Then we can start developing the Hyperty Reporter of the previously defined Hello Hyperty Object. First the constructor where we are going to instantiate the Hyperty syncher. For that, we take as input parameter the Hyperty address (HypertyURL), the MiniBus used to receive and send data synchronization messages (bus), and any required configuration data.

constructor(hypertyURL, bus, configuration) {  
  
 let syncher = new Syncher(hypertyURL, bus, configuration);  
}

1. Then we prepare the creation of the Hyperty Hello Data Object by defining the Catalogue URL from where the previously defined Hyperty Data Object schema can be retrieved and initial data to be used in the creation:

\_this.\_objectDescriptorURL = 'hyperty-catalogue://example.com/.well-known/dataschemas/HelloWorldDataSchema';  
  
let hello = {  
 hello : "Hello World!!"  
};

1. The object is created and another Hyperty is invited to be an observer.

syncher.create(\_this.\_objectDescriptorURL, [invitedHypertyURL], hello).then(function(dataObjectReporter) {  
  
 console.info('Hello Data Object was created: ', dataObjectReporter);

1. From the Observer side, in order to observe the Hello Data Object, it has to subscribe it, by passing its URL and the Catalogue URL from where its schema can be retrieved. As soon as the subscription is accepted the most updated version of the Hello Data Object is received ...

syncher.subscribe(\_this.\_objectDescriptorURL, ObjectUrl).then(function(dataObjectObserver) {}  
  
 console.info( dataObjectObserver);

1. Any change performed by the Reporter in the the object.

dataObjectReporter.data.hello = "Bye!!";

1. ... will be immediately received by the Observer:

dataObjectObserver.onChange('\*', function(event) {  
 // Hello World Object was changed  
 console.info(dataObjectObserver);  
 });

1. Define the Hyperty descritor. The most important attributes are:

* constraints to be fullfilled by the Runtime, in this case the only constraint is to be executed in a browser: browser: true
* hypertyType or the schemes of the data objet: "hypertyType": ["hello"]
* the catalogue URL of the data object schemas supported by the Hyperty, in this case: "dataObjects": ["https://%domain%/.well-known/dataschema/HelloWorldDataSchema"]

The complete descriptor is:

{  
 "language": "javascript",  
 "signature": "",  
 "configuration": {},  
 "constraints": {  
 "browser": true  
 },  
 "hypertyType": [  
 "hello"  
 ],  
 "dataObjects": [  
 "https://%domain%/.well-known/dataschema/HelloWorldDataSchema"  
 ]  
}

1. develop an App to test your Hyperty by using the [template used in "dev-hyperty" repo](https://github.com/reTHINK-project/dev-hyperty/tree/master/examples/hello-world).
2. test and run your Hyperty,

start the toolkit for browser:

npm run start:browser

Open https://catalogue.localhost/ and accept certificate

Open https://localhost/ and select your Hyperty to be tested

Any changes performed in your Hyperty source code will be automaticaly loaded by the toolkit to be tested.