RENDERJS

RenderJS - Homepage

Sven Franck, Ayush Tiwari

Table of Contents

Figure-1 - RenderJS Logo

1. RenderJS

RenderJS is a fully promise-based JavaScript library for building single-page web applications from reuseable components, called gadgets. It is developed and maintained by Nexedi and used for the responsive ERP5 interface and basis for applications in app stores like OfficeJS.

2. What are Gadgets?

Gadgets consist of HTML+JS+CSS and should ideally function standalone or within a tree structure of gadgets making up an application. Gadgets can be sandboxed (loaded inside iframe) or placed in the DOM. They communicate with each other by declaring and publishing methods (to make them accessible to child gadget) as well as acquiring them (from any parent gadget). At Nexedi, RenderJS is usually used in combination with jIO allowing to build complex applications connecting with multiple storages in a non-obstrusive, verbose and maintainable way.

3. Promises

RenderJS is fully asynchronous and use promises provided by a library called RSVP (original version, renderJS custom version below!). The main difference to the official Promise spec and RSVP is that RenderJS is not using **.then** for chaining. Instead chains are written using **RSVP.Queue()** with **.push(function () {...}).push(function (result) {...}).push(undefined, function (error) {...});** as custom extension which allows RenderJS promises to be cancelled.

4. Another JavaScript Framework? Why use RenderJS?

Nexedi's free software products and custom solutions developed from them are normally running for many years. As complexity of apps is usually very high, redevelopments to follow the current trending JS-framework or having to replace a framework being discontinued is out of our scope. Hence RenderJS (and jIO), two non-frills libraries, that are:

- **sturdy**, small API, easy to use once understood.
- **maintainable**, small number of resuable components.
- **controllable**, the full app as a single, cancellable chain of gadgets and promises.
- **configurable**, build anything, wrap anything, promisify anything.
- **cross-domain**, embed 3rd-party gadgets within any application

5. Getting Started

RenderJS is easy to setup and get working.

5.1. Source Code

The RenderJS source code is available on Gitlab (Github Mirror). To **build**,

```
> git clone https://lab.nexedi.com/nexedi/renderjs.git
> npm install
> grunt server
```

or just **download** the files directly:

- RenderJS latest version
- RenderJS latest version (minified)
- RSVP (custom version) (amd version)
- RSVP (custom version/minified)

The following file(s) might also be useful:

- gadget_global.js (promisified event bindings and file readers)
- jIO latest version
- jIO latest version (minified)

5.2. Hello World

Create an html and js file, with the following contents:

```
<!-- gadget_hello.html -->
<!Doctype html>
<html>
 <head>
  <title>Hello World Gadget</title>
  <script type="text/javascript" src="rsvp.js"></script>
  <script type="text/javascript" src="renderjs.js"></script>
  <!--<script type="text/javascript" src="gadget_hello.js"></script>-->
 </head>
 <body>Hello World</body>
</html>
```

In theory, there is nothing more to do. You can put HTML content inside a gadget. To do the same dynamically, uncomment the script above to load the file below.

```
/* gadget_hello.js */
/*jslint nomen: true, indent: 2, maxerr: 3 */
/*global window, rJS, RSVP */
(function (window, rJS, RSVP) {
 "use strict";

 rJS(window)

  .declareMethod('render', function () {
   this.element.textContent = "Hello World";
  });

}(window, rJS, RSVP));
```

Load your gadget file and it should display the *Hello World* text in the browser. Keep reading to find out what else you can do.

6. API - Quickguide

Below is a list of all methods provdided by renderJS followed by a more detailed explanation in the next sections.

Do what?	Do this!	Explanation
Declare Gadget (HTML)		The *url* is requrired, you can optional *scope* to retrieve gadgets in JavaScript, sandbox can be "public" (wrap gadget in <iv> or "iframe" (wrap gadget in <iframe>).
Declare Gadget (JS)	this.declareGadget("path_to_gadget.html", { "scope": "foo-1", "sandbox": "public", "element": this.element.querySelector(".foo_wrap") });	[returns Promise]. Same as in HTML, additional parameters include "element", where you can specify an element to be used as the gadget wrapper (instead of the).
Set Initial State	[rJS].setState({value: ''})	[returns Promise]. Gadget state should be set once when initialising the gadget. State should contain key/value pairs, but there is no hard restriction on what can be put in state.
Change State	this.changeState({"value": 123});	[returns Promise]. Change state by passing a new key-value, which will **only** overwrite the keys provided in the *changeState* call. Other keys will remain unchanged.

Do what?	Do this!	Explanation

Change State Callback	[rjs].onStateChange(function (modification_dict) {
	if (modification_dict.hasOwnProperty("value") {
	// do something
	}
	})
		[returns Promise]. Trigger fired whenever gadget state changes. Passes *modification_dict* which includes all modified state parameters.

Ready Handler	[rjs].ready(function () {
	// all dependencies loaded, do initialising things
	})
		The *ready* handler is triggered automatically when all gadget dependencies have loaded.

Render Handler	[rjs].declareMethod("render", "function (options) {
	// manual initialiser
	})
		render is the manual handler to initialise a gadget and is usually called from a parent gadget or in a *declareService* call. Initialisation parameters are usually passed into a gadget through render. However it is not necessary to have a *render* method although it is good practice so other gadgets have a common way to access a child gadget.

Declaring Methods	[rjs].declareMethod("name", function (params) {
	// method code
	})
		[returns Promise]. Declaring methods is the most common way of adding functionality to a gadget. Only declare methods which require the *this* context and/or should be accessible by parent/child gadgets.

Declaring Services	[rjs].declareService(function (params) {
	// method code
	})
		[returns Promise].*declareService* handlers will trigger once the gadget DOM is on the page and is ususally used for event bindings. There can be multiple *declareService* handlers, which trigger simulataneously.

Do what?	Do this!	Explanation

Declaring Jobs	[rjs].declareJob(function (params) {
	// method code
	})
		[returns Promise].*declareJob* handlers are an alternative to declaring services. They trigger as soon as possible and should be used instead of declaring services.

Event Binding	[rjs].onEvent("foo", function (event) {
	// method code
	}, false, true)
		[returns Promise]. The *onEvent* handler is set on the gadget element and captures all bubbling events. Use it to declare form submit bindings, etc. Alternatively you can use the *loopEventListener* and *promiseEventListener* definded in gadget_global.js and explained below.

Acquiring Methods	[rjs].declareAcquiredMethod('methodThisGadgetWantsFromAParent', 'methodThisGadgetWantsFromAParent')	Use this handler to acquire methods from parent gadgets passing the method to acquire in the first parameter and the name to call it locally (gadget.[....]) as second parameter.

Publishing Method	[rjs].allowPublicAcquisition(
	'nameToAcquire',
	function (param) {
	// call local method
	})
		In order for a method to be acquired, a parent gadget has to publish it as shown here.

7. API - Detailed Explanation

The following section will explain the RenderJS API in detail using two more detailled files.

7.1. HTML Gadget

A typical gadget would look something like this.

```
<!Doctype html>
<html>
 <head>
  <title>Example Gadget</title>

  <link rel="stylesheet" href="gadget_some.css" />

  <script type="text/javascript" src="rsvp.js"></script>
  <script type="text/javascript" src="renderjs.js"></script>

  <script type="text/javascript" src="gadget_global.js"></script>
  <script type="text/javascript" src="gadget_example.js"></script>
 </head>
 <body>
  <h1>Example Title</h1>
  <section></section>
  <div data-gadget-url="gadget_do_something.html"
     data-gadget-scope="do_something"
     data-gadget-sandbox="public">
  </div>
 </body>
</html>
```

Note all gadgets require to have all dependencies declared explicitly (because gadgets should work standalone). RenderJS will prevent depedencies from being loaded multiple times.

The above gadget uses a *gadget_global.js* which contains common methods for "promisifying"

events and readers. The *gadget_example.js* is discussed in detail below.

In this example there is single child gadget defined called *do_something* (you can define as many gadgets as you like). The scope is used internally in RenderJS to reference the gadget, for example when you want to call its *render* method. If you don't need to adress a gadget specifically from JavaScript, you can also omit the scope and let renderJS assign one for internal use only.

The *sandbox* parameter can be set to *public* causing all gadgets to be placed directly into the DOM or *iframe* resulting a gadget to be loaded inside an iFrame. This is meant to be used for embedding 3rd-party gadgets into an application where it can be desirable to restrict access these gadgets have with the application they are embedded in.

The example above includes some HTML content. This is not necessary and usually the gadgets being loaded are adding most of the content to the DOM, which is done automatically by RenderJS as the gadget is being rendered.

7.2. JavaScript Gadget

Below is a gadget using the full API explained step by step in detail (full code at the end).

```
/*jslint nomen: true, indent: 2, maxerr: 3 */
/*global window, document, rJS, RSVP, loopEventListener, promiseEventListener */
(function (window, document, rJS, RSVP, loopEventListener, promiseEventListener) {
 "use strict";

 ////////////////////////////////
 // some variables
 ////////////////////////////////
 var NOT_USED = "abc123";

 ////////////////////////////////
 // some methods
 ////////////////////////////////
 function checkChange() {
  var gadget = this;
  return gadget.changeState({
   value: gadget.element.querySelector('input[type="text"]').value
  });
 }

 function createForm(param) {
  var fragment = document.createDocumentFragment(),
   form = document.createElement("form"),
   input = document.createElement("input"),
   submit = document.createElement("input");
```



```
  form.setAttribute("name", "foo");
  input.setAttribute("type", "text");
  input.setAttribute("value", param);
  submit.setAttribute("type", "submit");
  submit.setAttribute("value="Submit");
  form.appendChild(input);
  form.appendChild(submit);
  fragment.appendChild(form);
  return fragment;
}

rJS(window)
```

Every gadget usually starts with some variable declarations and methods which do not have to be published on the gadget itself (think of internal parameters and methods). The method *checkChange* above is the callback run on detection of input events explained below. It retrieves a text input's value and triggers a state change with this value. States are explained in detail below as well. *createForm* assembles a HTML form element using the parameter provided.

The last part in this snippet is the RenderJS object **rJS(window)**. It's only used internally but has to be at the start of every gadget chain. Of course, the whole gadget must always be wrapped in a closure passing the globals being accessed.

```
  ////////////////////////////////
  // state
  ////////////////////////////////
  .setState({value: ''})
```

Gadgets are "state-able". The state is a dictionary of parameters, which is initialized using the **setState({"foo":"bar"})** method. Above the state is initialized with just a single key, called value, which will be updated by the *chechChankge* method shown in the previous snippet.

You can change state using the **changeState({"baz":"bam"})** method. This will only update/add the parameters passed, so in the current example, the key *baz* would be added to the state and if you would call **gadget.state**, you would receive **{"foo": "bar", "baz": "bam"}**. State changes can be handled using the **.onStateChange(modification_dict) {...}** handler shown below, which will pass a dict with only the updated state parameters. Let's continue.

```
  ////////////////////////////////
  // ready
  ////////////////////////////////
  .ready(function () {
    var gadget = this;
```



```
  console.log("READY - dependencies loaded");

  return new RSVP.Queue()
    .push(function () {
      return gadget.changeState({"counter": 123});
    })
    .push(function ()
      console.log("READY - gadget configuration");
      console.log(gadget.state);
      console.log(gadget.element);
    ]);
})
```

.ready triggers once all dependencies of a gadget as well as all gadgets declared in HTML have been loaded (gadgets have not been initialized, this is done using a gadget's **render** method).

You can think of it similar as jQuery *$(document).ready*. You can have multiple **.ready** calls, but they all fire in parallel, so you cannot declare a parameter in one call and expect it to be available in another.

Older examples of renderJS may still use the *getElement* method to retrieve the gadget element and store it on the gadget. This has been automated and you can access the gadget DOM using **gadget.element** directly.

The above snippet just shows the use of the *changeState* method by adding another parameter to the state dict, called *counter*. This will trigger the *onStateChange* method declared further down. The *ready* handler closes with showing how to access a gadget's state and (DOM) element.

```
////////////////////////////////
// acquired methods (from parent gadgets)
////////////////////////////////
.declareAcquiredMethod('methodThisGadgetWantsFromAParent', 'methodThisGadgetWantsFromAParent')
```

Gadgets can requests methods published by any parent gadget using **.declareAcquiredMethod**. Imagine you have a gadget that handles access to a storage or server. Only this gadget should interact with the server directly and publish it's methods so they are available to all other gadgets who need to request items from the server. Method declaration and acquisition allows to clearly separate functional logic, so a change in storage would mean only having to modify the storage gadget. **Note** that methods can only be acquired from gadgets higher up in the gadget tree (parents). Let's assume the method above just increments the parameter passed by 1.

```
////////////////////////////////
// published methods (to child gadgets)
////////////////////////////////
.allowPublicAcquisition('methodThisGadgetWantsFromAParentToChildren', function (param) {
  return createForm(param);
```



```
})
```

The pendant to acquiring a method is publishing a method, thereby making it accessible to be called from all child gadgets. To stay with the storage example, it would probably be wise to place a storage gadget relatively high up on the gadget tree in order to make sure that all child gadgets who need access can actually access the storage.

```
////////////////////////////
// state change
////////////////////////////
.onStateChange(function (modification_dict) {
  var gadget = this,
    input = gadget.element.querySelector('input[type="text"]'");

  console.log("state change, modifcation = ", modification_dict);

  if (modification_dict.hasOwnProperty("value") {
    // input.value = this.state.value;
    input.value = modification_dict.value;
  }
})
```

As mentioned above, any change in state will trigger the *onStateChange* handler with the modification_dict including just the state parameters that have been changed. In the example, the changeState which introduced the *counter* key will trigger the *onStateChangeHandler* but the method looks for the *value* key only which has not changed hence it will not be in the *modification_dict* and the method will do nothing.

```
////////////////////////////
// declared methods
////////////////////////////
.declareMethod('render', function (my_option_dict) {
  var gadget = this;

  console.log("RENDER - called automatically or through parent gadget");
```

Declared methods contain all methods a gadget is using or wants to expose to other gadgets. Other methods can remain "internal" and are placed outside of the rJS chain. A gadget should always declare a **render**, which allows a parent gadget to trigger rendering of this gadget without knowing what the gadget actually does while also being able to pass in parameters (*option_dict*). If one can assume that all gadgets have a *render* method, it would be easy to access a gadget and for example make it expose it's api. The idea is also that gadgets need not know about each other and their functionalities. *render* is the common entry point all gadgets share.

```
  return new RSVP.Queue()
```



```
// intialize the gadget already declared in HTML
.push(function () {
 // only one promise in parallel execution, could be more
 return RSVP.all([
  gadget.getDeclaredGadget("do_something")
 ]);
})
.push(function (my_gadget_list) {
 return RSVP.all([
  my_gadget_list[0].render(option_dict)},
 ]);
})
```

In this section we start a **RSVP.Queue()** chain which is what most gadgets contain. It is possible to originate queues from gadget based methods directly, so the separate call to **RSVP.Queue().push(function () {return first_method();})**... isn't necessary, because you can **.push()** on *first_method* directly.

The chain above shows how to access a gadget declared in HTML. Doing it this way requires the gadget declared in the HTML file to have an explicit scope so it is possible to reference this gadget from the gadget JavaScript. As described earlier, declaring in HTML means all dependencies will have been loaded when the *.ready* event fires. You can also declare gadgets directly inside JavaScript. This is shown further down. Once the gadget is available, we call it's *render* method passing in the configuration received. This is a common pattern of handing things such as configuration information from gadget to gadget.

Note that *RSVP.all([...])* is only used for demonstration purpose here, as there is only a single Promise to be returned. But you could also do the same process for multiple independent gadgets in parallel by adding their respective getDeclaredGadget and render calls to the **RSVP.all([...])** promise list.

```
// call the method needed and pass the counter
// assume it increase the counter by 1
.push(function () {
 return gadget.methodThisGadgetWantsFromAParent(gadget.state.counter);
))

// call gadget internal (and published) method with the result
.push(function (result) {
 return gadget.methodThisGadgetWantsFromAParentToChildren(result);
})

// add the form to the DOM
.push(function (content) {
 var div = document.createElement("div");
 div.appendChild(content);
 gadget.element.appendChild(div);
```



```
  // change state to update input field in case a value was passed in option_dict
  return gadget.changeState({value: option_dict.value});
})
```

Next we call the method we set to be acquired from a parent gadget calling it with one of the parameters defined on the gadget state dict. Acquired methods can be called directly on the gadget context and will always return a promise. In case the acquisition failed for some reason, an error will be thrown. In the snippet above we assume the acquired method to increase the counter passed in by 1 and returning it to the next step.

In this step, the result of the called method is used in a method this gadget publishes. If the gadegt itself only exposes a method to other gadgets, the method can directly be added to the **allowPublicAcquisition** handler like so:

```
/*
//////////////////////////////
// published methods (to child gadgets)
//////////////////////////////
.allowPublicAcquistion('methodThisGadgetWantsFromAParentToChildren', function (my_parameter) {
 return my_parameter;
}
*/
```

Of course, this method will then not be available on the gadget itself.

In the above example the returned counter is passed into the published method which returns an HTML string containing a form. This is added to the DOM in the next step before another *changeState* is triggered, updating our state value with the value passed in from the parent gadget upon initialization. **Note**, that this time *onStateChange* will actually try to do something as the modification dict will include the changed value parameter.

```
// declare another gadget dynamically, using declareJob
.push(function () {
 return gadget.deferRenderGadget();
})
```

The next snippet will show how to declare a gadget dynamically from within JavaScript- The method *deferRenderGadget* was created using **declareJob** (shown further below). Jobs are a way to postpone something to the earliest possible moment. In previous version of renderJS this function was not available often causing many methods having to be called on declareService (imagine loading a table and having to wait for table headers to load table content). DeclareService will "prepone" the job to be run as soons as possible causing less interruptions in the UI.

Note that calling a job here will post it to be executed as soon as possible but NOT within this promise chain. The code will immediately jump to the next step and any errors caused from the job will not show up in the error handler of this chain. Instead they will be thrown in the error handling set in *declareService*.

```
    .push(undefined, function (my_error) {
      console.log(my_error);
      throw my_error;
    });
})
```

The last step in the chain traps any errors and throws them. In theory a single error handler in the intial gadget is enough if you can ensure a promise chain is never broken throughout an application and all event listeners are properly wrapped in promises. In this case, the error will be propagated to this top-most error handler. It is however good practice to add error handlers on important methods to ensure that errors are traceable even if a chain is broken.

```
////////////////////////////////
// gadget event binding
////////////////////////////////
.onEvent('change', checkChange, false, true)
.onEvent('input', checkChange, false, true)
```

There are two way of event binding in RenderJS. You can bind to the gadget (similar to the document) or to elements directly (shown below). Binding to the gadget can be done using the **onEvent** handler, specifying the event to listen to, the callback declared initially and the *useCapture* and *preventDefault* parameters.

```
////////////////////////////////
// declareJob
////////////////////////////////
.declareJob('deferDeclareGadget', function () {
  var gadget = this,
    element = gadget.element,
    div = document.createElement('div');

  element.appendChild(div);

  return new RSVP.Queue()
    .push(function () {
      return gadget.declareGadget("gadget_do_something_else.html", {
        scope: "do_something_else",
        element: div
      })
    })
    .push(function () {
      return gadget.deferRenderGadget();
    });
})
```



```
.declareJob('deferRenderGadget', function () {
  var gadget = this;
  return gadget.getDeclaredGadget("do_something_else")
    .push(function (my_gadget) {
      return my_gadget.render({
        "other": "parameters",
        "to": "pass"
      });
    });
})
```

The next snippet specifies that jobs to run as fast as possible. In the above case the first job is declaring a gadget dynamically in JavaScript (**Note** that you can pass in an element which any HTML this gadget generates will be nested in). The job is finished by calling another job implying that this method will run as soon as the previous method has finished, in this case once the gadget and all dependencies have been loaded.

The second job retrieves the declared gadget, which is only possible once the previous job finishes and calls that gadgets *render* method, passing in a different set of parameters.

```
////////////////////////////////
// DOM element event binding
////////////////////////////////
.declareService(function () {
  var gadget = this,
    props = gadget.property_dict,
    form = props.element.querySelector("form");

  console.log("DECLARESERVICE - content available in DOM");
```

The next section is called **declaredService** and handles DOM element binding. It triggers once the underlying gadgets DOM has been built (compared to *ready* firing once dependencies have been loaded and *render* being initialized through the parent gadget. Imagine a graph library requiring the available width on screen to render a graph - this cannot be done on *ready* or *render*, because the gadget is available only in memory at this time. Once the DOM is built, all declareService(s) trigger (there can be more than one, too). Another option of doing this would be *declareJob* of course.

Note that *querySelector(All)* is the preferred way of querying the gadget HTML, because it is also available while a gadget is still being rendered. **Note** also, that it is not allowed to use *id* attributes anywhere in a gadget, because you cannot prevent multiple gadgets from existing on the same page. The gadget *scope* parameter must instead be used to access a gadget. For example, if you want to create two storage instance from the same gadget, you can do so like this:


```
/*
<div data-gadget-url="gadget_jio.html"
   data-gadget-scope="jio_gadget_localstorage"
   data-gadget-sandbox="public">
</div>
<div data-gadget-url="gadget_jio.html"
   data-gadget-scope="jio_gadget_webdav"
   data-gadget-sandbox="public">
</div>
*/
```

The gadget itself will only be loaded once, but two instances of the gadget will be available and adressable by their respective scope.

```
  function callback(my_event) {
    console.log("form submit registered");
    my_event.preventDefault();
    return false;
  }

  // form submit binding
  if (form) {
    return loopEventListener(form, "submit", false, callback);

  // example showing single-use promiseEventListener
  } else {
    return new RSVP.Queue()
      .push(function () {
        var button = "<button>Single Use Button</button>";
        props.element.appendChild(button);
        return promiseEventListener(props.elemnt.querySelector("button", "click", true);
      })
      .push(function (my_event) {
        alert(my_event);
      });
  }
});
```

This section shows the principles of wrapping event bindings and form events inside promises. The *gadget_global.js* provides the underlying methods, called **loopEventListener** (can trigger multiple times) and **promiseEventListener** (triggers a single time). **Note** the loopEventListener is also used in the *onEvent* handler.

The **loopEventListener** is set on the form submit and calls the defined callback. The **promiseEventListener** does not have a callback, instead it just jumps to the next step in the promise chain. It is a single-use promise, so it should not be used to bind to interactive elements such as buttons.

```
}(window, document, rJS, RSVP, loopEventListener, promiseEventListener));
```

Close by passing in the necessary global parameters.

This example covers everything you can do with RenderJS. The full code can be found below as well as further information and examples.

```
/*jslint nomen: true, indent: 2, maxerr: 3 */
/*global window, document, rJS, RSVP, loopEventListener, promiseEventListener */
(function (window, document, rJS, RSVP, loopEventListener, promiseEventListener) {
 "use strict";

 ////////////////////////////
 // some variables
 ////////////////////////////
 var NOT_USED = "abc123";

 ////////////////////////////
 // some methods
 ////////////////////////////
 function checkChange() {
  var gadget = this;
  return gadget.changeState({
   value: gadget.element.querySelector('input[type="text"]').value
  });
 }

 function createForm(param) {
  var fragment = document.createDocumentFragment(),
   form = document.createElement("form"),
   input = document.createElement("input"),
   submit = document.createElement("input");

  form.setAttribute("name", "foo");
  input.setAttribute("type", "text);
  input.setAttribute("value", parameter);
  submit.setAttribute("type", "submit");
  submit.setAttribute("value="Submit");
  form.appendChild(input);
  form.appendChild(submit);
  fragment.appendChild(form);
  return fragment;
 }

 rJS(window)

  ////////////////////////////
  // state
  ////////////////////////////
  .setState({value: ''})

  ////////////////////////////
  // ready
  ////////////////////////////
  .ready(function () {
   var gadget = this;

   console.log("READY - dependencies loaded");

   return new RSVP.Queue()
    .push(function () {
     return gadget.changeState({"counter": 123});
    })
```



```
    .push(function ()
      console.log("READY - gadget configuration");
      console.log(gadget.state);
      console.log(gadget.element);
    ]);
  })

  ////////////////////////////////
  // acquired methods (from parent gadgets)
  ////////////////////////////////
  .declareAcquiredMethod('methodThisGadgetWantsFromAParentToChildren', 'methodThisGadgetWantsFromAParent')

  ////////////////////////////////
  // published methods (to child gadgets)
  ////////////////////////////////
  .allowPublicAcquistion('methodThisGadgetWantsFromAParentToChildren', function (param) {
   return createForm(param);
  })

  ////////////////////////////////
  // state change
  ////////////////////////////////
  .onStateChange(function (modification_dict) {
   var gadget = this,
    input = gadget.element.querySelector('input[type="text"]"');

   console.log("state change, modifcation = ", modification_dict);

   if (modification_dict.hasOwnProperty("value") {
    // input.value = this.state.value;
    input.value = modification_dict.value;
   }
  })

  ////////////////////////////////
  // declared methods
  ////////////////////////////////
  .declareMethod('render', function (option_dict) {
   var gadget = this;

   console.log("RENDER - called automatically or through parent gadget");

   return new RSVP.Queue()

    // intialize the gadget already declared in HTML
    .push(function () {
     // only one promise in parallel execution, could be more
     return RSVP.all([
      gadget.getDeclaredGadget("do_something")
     ]);
    })
    .push(function (my_gadget_list) {
     return RSVP.all([
      my_gadget_list[0].render(option_dict)},
     ]);
    })

    // call the method needed and pass the counter
    // assume it increase the counter by 1
    .push(function () {
     return gadget.methodThisGadgetWantsFromAParent(gadget.state.counter);
    ))
```



```
    // call gadget internal (and published) method with the result
    .push(function (result) {
      return gadget.methodThisGadgetWantsFromAParentToChildren(result);
    })

    // add the form to the DOM
    .push(function (content) {
      var div = document.createElement("div");
      div.appendChild(content);
      gadget.element.appendChild(div);

      // change state to update input field in case a value was passed in option_dict
      return gadget.changeState({value: option_dict.value});
    })

    // declare another gadget dynamically, using declareJob
    .push(function () {
      return gadget.deferRenderGadget();
    })

    // capture errors
    .push(undefined, function (my_error) {
      console.log(my_error);
      throw my_error;
    });
}

////////////////////////////////
// gadget event binding
////////////////////////////////
.onEvent('change', checkChange, false, true)
.onEvent('input', checkChange, false, true)

////////////////////////////////
// declareJob
////////////////////////////////
.declareJob('deferDeclareGadget', function () {
  var gadget = this,
    element = gadget.element,
    div = document.createElement('div');

  element.appendChild(div);

  return new RSVP.Queue()
    .push(function () {
      return gadget.declareGadget("gadget_do_something_else.html", {
        scope: "do_something_else",
        element: div
      })
    })
    .push(function () {
      return gadget.deferRenderGadget();
    });
})

.declareJob('deferRenderGadget', function () {
  var gadget = this;
  return gadget.getDeclaredGadget("do_something_else")
    .push(function (my_gadget) {
      return my_gadget.render({
        "other": "parameters",
```



```
        "to": "pass"
      });
    });
  })

  ////////////////////////////////
  // DOM element event binding
  ////////////////////////////////
  .declareService(function () {
   var gadget = this,
    props = gadget.property_dict,
    form = props.element.querySelector("form");

   console.log("DECLARESERVICE - content available in DOM");

   function callback(my_event) {
    console.log("form submit registered");
    my_event.preventDefault();
    return false;
   }

   // form submit binding
   if (form) {
    return loopEventListener(form, "submit", false, callback);

   // example showing single-use promiseEventListener
   } else {
    return new RSVP.Queue()
     .push(function () {
      var button = "<button>Single Use Button</button>";
      props.element.appendChild(button);
      return promiseEventListener(props.elemnt.querySelector("button", "click", true);
     })
     .push(function (my_event) {
      alert(my_event);
     });
   }
  });

}(window, document, rJS, RSVP, loopEventListener, promiseEventListener));
```


8. Tips and Tricks

8.1. Skip Queues

It is not necessary to explicitly start all chains with a call to **return new RSVP.Queue()** as all methods available on *gadget* are queue-able. In the example it is done for demonstration purposes but the following is also possible:

```
/* Bad example */
 return new RSVP.Queue()
  .push(function () {
   return gadget.changeState({"counter": 123});
  })
  .push(function ()
   console.log("READY - gadget configuration");
   console.log(gadget.state);
   console.log(gadget.element);
  ]);

 /* Good example */
 return gadget.changeState({"counter": 123})
  .push(function () {
   console.log("READY - gadget configuration");
   console.log(gadget.state);
   console.log(gadget.element);
  });
```

allowing to write more concise and maintainable code.

8.2. DeclareService Not Firing

While you should use *declareJob* or *.onEvent* in favor of *declareService* sometimes it is still useful when manually building parts of the DOM. As explained the declareService handlers will trigger when the gadgets DOM has been placed on the page. **Note** that gadget DOM means refers

to the full gadget, including it's ** ** wrapper. Just appending parts built inside the gadget will not trigger *declareService*

8.3. Only Use onEvent

The DOM element binding using *declareService*isn't really necessary, as you can easily capture events bubbling up from a DOM element to the containing gadget and handle them there. For example:

```
.onEvent('submit', function (event) {
  var target = event.target[0];
  if (target.getAttribute("name") === "some_form") {
    return handleThisSpecificForm(event);
  }
}, false, true);
```

Using this way of setting bindings on the gadget instead of the nested DOM element allows to write much easier code and bundles all bindings on the respective gadget.

8.4. Non-Bubbling Events

You can also use *onEvent* (set on the gadget wrapping ** ** element) for non-bubbling events like so:

```
//someEventThatDoesNotBubble
.onEvent("invalid", function (my_event) {
  return callbackFunction();
})
```

8.5. Ready is tricky

It is good practice to not do any gadget operations such as calling *render* or loading and working with sub-gadgets within *ready*. Do this when calling *render* manually. Also note you can have multiple *.ready* handlers but they will all fire in parallel, so on *.ready* handler cannot depend on the outcome of another.0

8.6. Error handling

The advantage of running an application as a single chain of promises is the ability to capture and handle errors. Imagine an application crashing for some reason, RenderJS being able to capture the error and sending an Ajax request with an error report to a log instead of an app just breaking. Basic error handling within JavaScript is already possible using the above error handler at the end of promise queues:

```
  return new RSVP.Queue()
    .push(function () {...})
    .push(undefined, function (my_error) {
      console.log(my_error);
      throw my_error;
    });
})
```

In addition RenderJS allows to trap Service errors as well as error resulting from errors in the HTML and loading of dependency files. To also be able to handle these types of errors occuring "outside" JavaScript, add:

```
.allowPublicAcquisition('reportGadgetDeclarationError', function (argument_list, scope) {
    // Do not crash the UI in case of wrongly configured gadget,
    // bad network, loading bug.
    this.state.rejected_dict[scope] = null;
    console.log(argument_list[0]);
  })

.allowPublicAcquisition('reportServiceError', function (argument_list) {
    // Do not crash the UI in case of gadget service error.
    // do something, for example
    console.log(argument_list[0]);
  })
```


9. Tests

You can run tests after installing and building RenderJS by opening the */test/* folder.

10. FAQ

Q: What browsers does RenderJS support?

A: RenderJS will work on fully html5 compliant browsers. Thus, RenderJS should work well with the latest version of Chrome and Firefox. IE is a stretch and Safari as well. Run the tests tto find out if your browser is supported.

11. Licence

RenderJS is an open-source library and is licensed under the LGPL license. More information on LGPL can be found here.

12. Examples

Most of the front end solutions created by Nexedi are based on RenderJS and jIO. For ideas and inspiration check out the following examples:

- OfficeJS - Office Productivity App Store (Chat client, task managers, various editors).