Taist

Release 0.2

February 12, 2017

Contents

Taist is a tool to add custom functionality to cloud applications

Taist allows to create **addons** - bundles of Javascript code that are injected to a target application's web page via our browser extension

CHAPTER 1

What you can do with Taist:

- **add new features**: custom fields to existing entities, whole new entities that interact with existing ones, custom actions and workflows
- **extend an existing workflow** - add new steps or condition checks
- **automate repeatedly executed tasks** - solve them with less number of human actions
- **manipulate several applications** - execute tasks using only one application instead of switching between several ones, copy-pasting data, etc.

CHAPTER 2

Why create addons with Taist

We aim to make addons development an important part of the whole cloud software industry, so we create Taist as a powerful platform that helps you in every aspect of the addon development.

2.1 Now:

- Taist SDK solves common development tasks like:
  - **storing data** (with different access levels)
  - **integrating** into the native UI of a target application
  - **communicating** with any external services
- **Taist platform** solves all the administrative tasks like addons deployment and updating, user management, etc.
- **Rapid development** is ensured with features like IDE integration or Github integration

2.2 In the future:

- **Marketplace** will allow to make money with addons: sell paid addons and get clients for custom ones
- **Maintenance system** will help you mange the whole addons lifecycle with tools like automated testing and error reporting

CHAPTER 3

Next steps:

- Read how Taist works
- Create a “Hello, world!” addon
- Learn why Taist is better than custom browser extensions or GreaseMonkey scripts

CHAPTER 4

Support

- just write us to tell what sucks or what you like or to ask any question you have

4.1 First steps

4.1.1 How it works

Addons

Taist uses addons - bundles of Javascript and CSS that are injected into the target web pages. Each addon is launched on a specific website and usually implements a single enhancement or feature.

Anybody who knows Javascript can create addons.

How it works

- When you visit any web page our browser extension detects whether you have any addons that should be launched on it
- If such addons exists, our extension injects them to the page when it starts loading

What can addons do:

- **As a part of page**: addon code is executed in the scope of the page and is considered by the browser as just another piece of that page, so an addon can: * **communicate with website server** without any additional authentication - all the auth cookies for its requests are sent automatically by the browser as if those requests were made by original code * **reuse original frontend code** - call and modify it in any way needed * **modify original DOM and CSS**
- **As Taist addon** it gets access to addons SDK that provides it useful features: * communication with any external server regardless of website restrictions * data storage * convenient ways of integration into the original UI and logic of a web page

4.1.2 Why Taist

Everything you can do with Taist you can technically do without it by creating custom browser extensions or Grease-Monkey scripts.

But you will have to solve lots of tasks that are already solved in Taist - creating, deploying and administering browser extension, creating backend for data storage and server logic, user provisioning, etc.

Taist reduces your development and maintenance costs on orders of magnitude by solving all the secondary tasks and allowing you to focus on business logic and giving tools to easily manage and maintain it.

Some of Taist advantages:

- no need to create and maintain browser extension - just plain Javascript code, easily deployed and maintained
- built-in tools for easy development, deployment and support of addons
- out of the box features like
  - data storage with different access levels - user-based, company-based
  - user provisioning
  - integration into target web applications UI
  - support of cross-origin requests and Content Security Policy

4.1.3 Hello, world!

The usual way is developing addon locally and then publishing it to our server. So we will start from creating the simplest addon locally.

1. Clone addon repository from Github

2. Install extension:

- Install our Chrome extension
- Click on the extension icon to log in

3. Activate addon:

- Click on extension icon again and then on *"Local addon development"*
- Follow instructions there to give file access permissions
- Open "Local addon development" page again
- Turn addon ON
- Press `Choose file` and a new tab with your file system will open; use it to find and choose the file `hello_world.manifest` in the addon repository folder

4. Success!

- Visit Github: the page background should become dark and a `Hello, world!` alert should appear
- open browser console - you should see several lines prefixed with `[Taist]` that report on successful addon launch, including addon's message `Hello, console!`
- You have created your first taist addon. Now learn how it works and how to add some useful functionality to it.

4.2 Developer's Guide

4.2.1 Addon's manifest

Every addon needs a manifest that contains all the info for Taist to load and launch the addon. It should be a JSON file with `.manifest` extension.

Addon's file structure

An addon can have any folder and file structure. Even several addons can be placed inside the same folder. You just need to ensure that all the file paths used in an addon's `manifest` are correct.

Manifest structure

A manifest example:

```
{
  "id": "google-drive-folder-descriptions",
  "js": "build/drive_folder_descriptions.js",
  "title": "Folder descriptions",
  "siteName": "Google Drive",
  "siteRegexp": "drive.google.com",
  "shortDescription": "Add a description to any folder that is displayed in the folders list and ins
  "css": "build/drive_folder_descriptions.css",
  "description": "./drive_folder_descriptions.md",
  "pages": null
}
```

Manifest description (**mandatory** fields are in bold):

- **id** - unique name of an addon. Should consist of lowercase letters, digits and hyphens. Usually contains target application's name and addon purpose.
- **js** - relative path to addon's Javascript file from the folder that contains the manifest
- **title** - addon title
- **siteName** - name of the target application or website. It is also used to group addons for the same applications in addons list
- **siteRegexp** - regular expression to match the domain name of a target website. Typical use cases:
  - **a single common domain** is used for all the customers of the application - just use plain text like `drive.google.com`
  - **a separate subdomain for every customer** is used - just add `.` in the beginning: `.someapp.com` will run on `customer1.someapp.com` and `customer2.someapp.com` but will skip the marketing website *someapp.com*
  - **in other cases** just use an appropriate regular expression
- **shortDescription** - Short addon description in plain text
- **css** - relative path to addon's CSS file
- **description** - relative path to a file with detailed addon's description. Markdown can be used in it.
- **pages** - array of entry points to limit a set of pages to launch an addon on. It is not set in the example above as it can be used only for non-single page applications.

4.2.2 Writing Javascript

As we have learned while creating "Hello, world!" addon, Javascript part of addon is given in separate `.js` file and looks like:

This is a typical structure of Javascript part - let's learn how it works.

Addon initialization

`.js` file should contain single top-level function - in our example it is named `init`, but it can have any other or no name at all. It is run when addon is applied to target page.

This function creates and enclosures all the addon logic but does not launch it yet. In our example, single function *doWork* is defined. Instead, it returns an object (`addonEntry` in our example) that contains another function in its field `start`.

Addon launch

The `start` function actually launches the addon - in our example it calls `doWork`.

Arguments of `start` function:

- `taistApi` is an object that provides access to all Taist features. All the features are described in other chapters of this guide. In `doWork` we see use of `log` function - it's a preferred way of Debug output.
- `entryPoint` is used for fine-tuning of what addon parts to launch on what pages of target site.

4.2.3 Choosing specific pages to launch addons on

You can limit addons launch to specific pages of a website.

Important: when are addons launched

Addons are launched on every DOM load - it means, after the user navigates to a new page or reloads the current one, or navigates back/forward in history to other page, or reopens a previously closed page. In all these cases a fresh new page is loaded in the browser and every addon that fits it is relaunched on it.

After an addon is launched on the page it will not be launched again (its `start` function will not be called). So **addons are not launched on**:

- *hash change* - when just hash changes, page's DOM is not reloaded; if addon was already launched on this page it remains active, so no need in injecting and launching it again.
- *programmatic manipulation with history* - if `window.history` object is modified programmatically without actual page change, previous addons also remain on the page so no need to launch them again.

So there are different approaches on managing addons for *Sites that reload pages during navigation* and *Fully dynamic sites* (usually single-page applications)

Sites that reload pages during navigation

For such sites you can use **entry points** to limit your addon to some specific pages and split its logic for different pages. Entry points are set as `pages` property in `.manifest` file.

Let's look at example:

`.js` file:

```
function() {

      function tasksLogic(){
        alert('user part');
      };

      function adminLogic(){
        alert('admin part');
      };

  return {
    start: function(utils_described_later, entryPoint) {
      if (entryPoint === 'tasks')
            tasksLogic();
      else
            adminLogic();
    }
  };
}
```

`.manifest` file:

```
{
  "pages": [
    {
      "path": "^(/\w+)/tasks",
      "entryPoint": "tasks"
    },
    {
      "path": "^/admin/",
      "entryPoint": "admin"
    }
  ]
}
```

Entry points

Entry points are contained in object `pages` - there are two records with fields:

- `path` - a string representation of a Regexp
- `entryPoint` - a string name of current entryPoint

On a new page load, entryPoints are iterated in the order they are given in config:

- the first entryPoint with `path` that fits `location.pathname` is chosen as resulting entryPoint and addon is applied - it's `start` function receives resulting entryPoint name as a second parameter.
- if no entryPoint fits, addon is not applied

Results:

url	addon is applied	entryPoint	alert output
somesaas.com/admin/	Yes	admin	admin part
somesaas.com/adminproject/	No	-	
somesaas.com/clientproject/tasks	Yes	tasks	tasks part
somesaas.com/tasks	No	-	
somesaas.com/admin/tasks	Yes	tasks	tasks part

Fully dynamic sites

For sites that you should design addons to be applied from the very beginning on any page. Then you can watch for hash change using `taistApi.hash`:

- `taistApi.hash.onChange(callback)` - runs `callback(newHash, oldHash)` when the hash part of the URL is changed.
- `taistApi.hash.when(regexp, callback)` - runs `callback()` every time when hash changes and fits `regexp`. `regexp` can be given as a string or a RegExp object.

4.2.4 Debugging and troubleshooting

Debug and error output

Please, use `taistApi.log(...)` and `taistApi.error(err)` instead of `console.log` and `console.error`:

- they add prefixes that help to identify your addon output, like `[Taist] [DEVELOPED_ADDON]`
- later they will be enhanced to send output to addon author if it is needed to get additional debug information, and to properly display errors for addon user

Additional features of `taistApi.error(err)`:

- it checks `err.halt` - if it is `true`, `new Error(err.message)` is thrown stopping addon execution
- in production mode it sends error info to Taist developers to help with addon troubleshooting

Troubleshooting

If you experience any problems or bugs during addon development, try checking:

- target page console: Taist reports all errors appeared while addon launch and work there, including errors in `taistApi` calls
- addon development page: if addon is just not applied to target page, check manifest settings, go to addon development page and switch on the addon one more time - maybe some warnings will appear;
- extension background page's console: if addon still does not launch, open Taist extension background page (from Chrome's extensions page) and check its console: Taist writes there when it applies addons to any page
- if nothing helps, send us a bug report - there's a link in our extension's popup

Optional errors processing

As all errors from `taistApi` calls are always processed by Taist first, processing them in addons is not obligatory and ignoring them can simplify addon's logic.

So there are two error processing modes regulated by flag `taistApi.haltOnError` (`false` by default):

1. (default) Pass errors to addons: `haltOnError = false`

In this mode after error is processed by Taist it is passed to addon's callback, and all callbacks look like

```
function callback(error, result) {
  if (error) {
    // additionally process error here
    ...
  }
  else {
    // process result
    ...
  }
}
```

2. Halt addon execution on error: `haltOnError = true`

In this mode error is not passed to addon's callbacks - uncaught Error is thrown instead. So callbacks are simpler and look like

```
function callback(result) {
  // process result
}
```

3. Setting processing mode

`taistApi.haltOnError` can be changed in any moment - each `taistApi` call will use its value that was actual just before the call was started

4.2.5 Writing CSS

Addons can contain additional styles which are applied automatically during addon launch:

- put relative path to the css file to the `css` property ob the **manifest**
- check it by hitting `Save` on local addon development page: if file is not found, corresponding warning will be displayed.

4.2.6 Storing data

Taist allows to store data in different ways:

User data locally

`taistApi.localStorage` allows to store data using `localStorage` - it's just a wrapper around `localStorage` that allows storing objects:

- `set(key, value)` - stores value in `localStorage`; can store objects
- `get(key)` - gets previously stored value;
- `delete(key)` - deletes value

User and company data on server

Data can also be stored on Taist server:

- user-level data is processed via `userData`
- company-level data (shared between all company members) is processed via `companyData`

`userData` and `companyData` have the same basic interface:

- `set(key, value, callback)` - stores `value` on Taist server
- `get(key, callback)` - gets previously stored data
- `delete(key, callback)` - deletes `value`; throws an error if there is no data stored with given `key`

params:

- `key`, `value` - scalar value or JSON-like object
- `function callback(error, result)` - `result` is set only for `get` method

Company data: working with parts of values

`companyData` also allows to change or retrieve parts of data:

- `getPart(key, partKey, callback)` - gets field `partKey` of an object previously stored with key = `key`
- `setPart(key, partKey, value, callback)` - sets field `partKey` of an object previously stored with key = `key`

Example:

```
taistApi.companyData.set('foo', {bar: 1, baz: '2'}, function(setErr){
  taistApi.companyData.getPart('foo', 'bar', function(getPartErr, part){

    taistApi.log('foo.bar =', part);

    taistApi.companyData.setPart('foo', 'baz', {newBaz: true}, function(setPartErr){

      taistApi.companyData.get('foo', function(getErr, whole){

        taistApi.log('foo =', whole);
      })
    })
  })
})

/*
output:

foo.bar = 1
foo = {"bar":1,"baz":{"newBaz":true}}
*/
```

4.2.7 Basic integration with original UI

`taistApi.wait` provides simple and powerful ways to integrate addon's UI into original website UI.

Waiting for specific elements to render

The most important and heavily used is `taistApi.wait.elementRender`:

`elementRender(elementSelector, callback)` - fires the `callback` every time when a new element appears that matches `elementSelector`. If such elements exist on the page already, `callback` will also be fired for each of them

params:

- `elementSelector` - either jQuery-like string selector or a function that returns jQuery elements: every time new element appears that can be found using this selector or function, callback is fired
- **`function callback($element)` - callback to fire; is fired separately for each new element that matches `elementSe`**
  `$element` - jquery object containing single DOM element

behaviour:

- `callback` is fired only once for each `$element`;
- if several `elementRender` calls were made, every `callback` whose `elementSelector` matches current `$element` is fired

It's very convenient for reacting on any UI change. Example:

```
taistApi.wait.elementRender('.foo', function(element){
  taistApi.log('string selector fired:', element.text());
});

taistApi.wait.elementRender(function(){return $('p')}, function(element){
  taistApi.log('function selector fired:', element.text());
});

setTimeout(function(){
  $('body').append('<span class="foo">1</span>');

  $('body').append('<p>2</p><p>3</p>');

  $('body').append('<p class="foo">4</p>');
}, 1000);


/*

output:

string selector fired: 1
string selector fired: 4
function selector fired:  2
function selector fired:  3
function selector fired:  4

*/
```

Intercepting original UI actions

Intercepting actions is pretty straightforward - usually it is just adding new event handlers. Example - intercepting button click:

```
taistApi.wait.elementRender('button.saveDeal', function(saveButton){
  saveButton.click(function(){
    if (!additionalDealDetailsFilled()) {
      $('span.saveDealError').text('Please, save additional deal details');

      //prevent original click handler firing
      return false;
    }
    else
      //allow original click handler firing and deal save
      return;
  })
});
```

General-purpose waiting functions

There's a bunch of general-purpose functions that can be also handy when integrating into UI:

`taistApi.wait.once(condition, callback)` - waits for `condition` to become `true` once and fires a callback;

`taistApi.wait.repeat(condition, callback)` - waits for `condition` to become `true`, fires a callback and waits again;

params:

- `function condition()` - calculates condition, returns `true` when condition is met
- `function callback()` - callback

`taistApi.wait.change(expression, callback)` - fires `callback` every time `expression` changes value

params:

- `function expression()` - function to calculate current expression value

use cases: watch for change of selected tab or selected item list

4.2.8 Communicating with third-party services

`taistApi.proxy` provides methods to communicate with any third-party service without struggling with Same-origin policy or Content Security Policy:

`taistApi.proxy.jQueryAjax(host, path, settings, callback(taistError, responseFromRemoteServer))` works like jQuery.ajax:

pseudocode of successful call:

```
taistApi.proxy.jQueryAjax = function(host, path, settings, callback) {
  settings.url = host + path;
  settings.success = function(data, textStatus, jqXHR){
    callback(null, {
      error: null,
      statusCode: jqXHR.status,
      headers: jqXHR.getAllResponseHeaders().split('\n'),
      body: jqXHR.responseText,
      result: data
    })
```

```
  };

  jQuery.ajax(settings);
};
```

params:

- `host` - protocol (optional, `http://` is used by default) + hostname + port, like `google.com` or `https://my-server.com:8000`
- `path` - rest of url, starting with `/`, like `/doodles/finder/2014/All%20doodles`
- `settings` - almost identical to `settings` parameter of *jquery.ajax*, except that you can set only JSON-convertable options (functions will be just skipped)
  - `async`, `cache`, `contents`, etc. - OK
  - `beforeSend`, `complete`, `success` and other functions will be ignored while stringifying it to pass to Taist extension
- `taistError` contains only internal Taist error, not error from remote server:
  - if remote server responds with error (in `responseFromRemoteServer.error`), `taistError` is still `null`
  - if `taistError` is given, `responseFromRemoteServer` is `null`
- `responseFromRemoteServer` is an object with fields:
  - `statusCode` - status code like `200` or `401`
  - `body` - plain response body
  - `result` - contains response body automatically parsed by jQuery according to `settings.dataType`;
    - if response body cannot be stringified using `JSON.stringify` (to pass it to addon from extension), `result` is `null`, and `responseFromRemoteServer.resultStringifyError` is set and contains error message; `body` is still defined
  - `error` - error message from remote server; if it is set, `result` is not set;
  - `headers` - array of response headers like `["Cache-Control: private", "Server: nginx/1.1.19"]`

examples:

```
logResults = function(error, result){
  console.log('error:', error);
  console.log('result:', result);
};

//successful call to JSON api
taistApi.proxy.jQueryAjax('https://api.github.com', '/users/taist/repos', {}, logResults);

/* output
error: null
result: Object {error: null, statusCode: 200, headers: Array[20], body: «{ ...} ", result: Array[2]}
*/

//unsuccessful call
taistApi.proxy.jQueryAjax('https://www.googleapis.com', '/blogger/v2/blogs/2000', {method: 'GET'}, lo

/* output
error: null
```

```
result: Object {error: "Forbidden", statusCode: 403, headers: Array[14], body: "{error": { ... }"
*/
```

4.2.9 Dependencies

jQuery

jQuery is always available for addons: if it is not included in original website code, it is injected before addons. It is available as `window.$` (or as `window.jQuery` if `$` is used by the original website code).

Other Javascript libraries

Now external libraries have to be bundled with the addon using tools like Browserify. We are preparing a 'bootstrap' addon that uses this approach.

Other addons

We are planning to introduce addon dependencies. For now you can export addon's functionality via global variables that can be accessed by other addons.

4.2.10 Publishing addons

You can publish your addon for other Taist users. There are several types of addons:

- **free** - can be used by anyone for free;
- **paid** - can be used by anyone by subscription - X dollars per user per month; contact us at anton@tai.st for more details
- **private** - can be used only by a single company specified by addon author; contact us at anton@tai.st for more details

Publishing for the first time

- Open "Local addon development" page from popup of Taist extension
- Press "Publish addon"; a form to create new addon will open
- Fill the **manifest url** field (a manifest and other source files should be available over http or https)
- Add screenshots and save
- Disable local addon
- Your published addon will appear in the addons list. Activate it and check how it works

Updating published addon

- Remember to disable current published version of addon while developing new version locally
- To update an addon, just select it from addons list and press *Save* button