tozti Documentation

Release 0.1

the tozti project

Feb 28, 2018

Contents

This project is part of our scholarship at the ENS de Lyon, specifically the software project course of the M1 of Foundation of Computer Science supervised by Eddy Caron with the help of Damien Reimert.

CHAPTER 1

User's Guide

This part of the documentation is directed towards associations and association members which want to use this project.

CHAPTER 2

Developer's Guide

This part of the documentation contains specifications and explainations about the inner workings of the project.

2.1 Quickstart

To start working on tozti you only need python3 and git (you should be able to install them from your package manager). Make sure you have at least python 3.5 (for the `async/await` syntax) and `setuptools` installed:

```
python3 --version
python3 -m ensurepip --upgrade  # may need root privileges
```

One good way to organize yourself is to create a `tozti` folder somewhere:

```
mkdir tozti && cd tozti
```

A good practice when working on python projects is to setup a virtual environnement (venv). A venv behaves much like a separated python installation: it has it's own separated list of installed packages so that when you are working on two projects that need different version of a specific package you just put them in different venvs and install different versions on each. For more informations see the `venv` module and **PEP 405**. You may create a venv named `venv` inside the `tozti` folder with:

```
python3 -m venv venv      # create it
source venv/bin/activate  # activate it
```

Now that you are inside the venv (you should see `(venv)` at the beginning of your prompt), the `pip` and `python` commands will be aliased to the ones from the venv. To deactivate it just issue `deactivate`. Now you can clone the repos, install them inside your venv and start tozti:

```
git clone git@github.com:tozti/tozti && cd tozti
pip install -r requirements.txt
python3 -m tozti dev  # from the root of the repo
```

Extensions are located inside the `extensions` folder. To build one, make sure you have the `npm` package, then type:

```
npm install     # only needed when you change package.json
npm run build   # build the *.vue files to browser-compatible javascript
```

2.2 Architecture

Tozti serves 3 main HTTP namespaces:

- `/static`: usual static files (javascript, css, images, etc)
- `/api`: REST endpoints to interact with the server
- anything else will be responded to with the main `index.html`

2.2.1 Extensions

The tozti core is really lightweight but it has the ability to load extensions. During the startup, the server will search for extensions in the `extensions` subfolder of the tozti repository root.

Directory structure and `server.py`

The tozti core is really lightweight but it has the ability to load extensions. For now, you only need to know that extension is a folder providing a python file (*server.py* or `server/__init__.py`), describing how the extension works on the server (its routes, which files must be included from the client). This file must contain a global variable `MANIFEST` that is a dictionary containing the following keys (any one being optional):

`name` The name of the extension, in lower-case and with dashes instead of spaces.

`includes` A list of css or js files that must be included in the main `index.html`. Usually you will put there `"main.js"` which contains the code to register or patch components. The file paths must be relative to the `dist` subfolder of the extension (see below).

`_god_mode` Beware, this can be dangerous if used incorrectly! This should be a function taking as argument the main `aiohttp.web.Application` object. You can use it to register custom middlewares or do otherwise weird stuff.

The extension can contain a `dist` folder. The content of this folder will be served at the URL `/static/<extension-name>`.

Vuejs initialization

- See example in branch sample-extension.
- See an intro and some doc on components.
- See template syntax.

2.3 API

The tozti core provides an API to perform operations on the database prefixed with `/api/store`. This API is largely inspired by JSON API so you are encouraged to go take a look at their specification.

2.3.1 Error format

The format of the errors follows JSON API errors. If a request raised an error, the server will send back a response with status code `500`, `404`, `406` `409` or `400`. This response might send back a json object with an entry `errors` containing a list of json objects with the following properties:

`code` The name of the error

`status` Status code of the error

`title` Short description of the error

`detail` More about this error. This entry might not be present.

`traceback` Traceback of the error. This entry might not be present and is included only if tozti is launched in dev mode.

2.3.2 Concepts and Data Structures

Resources

Resources and resource objects are the main concept of the store API. A resource is what we would call an entity in SQL or hypermedia on the web. A *resource object* is represented as a json object with the following properties:

`id` An UUIDv4 which uniquely identifies a resource.

`href` A URL to the object itself.

`type` The name of a *type object*.

`body` A JSON object where the keys are strings and values are either *relationship objects* or arbitrary JSON value (ie *attributes*).

`meta` A JSON object containing some metadata about the resource. For now it only contains `created` and `last-modified` which are two self-explanatory dates in ISO 8601 format (UTC time zone).

Relationships

A relationship is a way to create a directed and tagged link between two resources. Relationships can be *to-one* (resp. *to-many*) in which case they link to one (resp. a sequence) of other resources. Practically, a *resource object* is a JSON object with the following properties (beware, here we diverge a little from the JSON API spec):

`self` An URL pointing to the current relationship object. This URL can be used to operate on this relationship.

`data` In the case of a *to-one* relationship, this is a *linkage object*, in the case of a *to-many* relationship, this is an array of *linkage objects*.

Linkages are simply pointers to a resource. They are JSON objects with three properties:

`id` The ID of the target resource.

`type` The type of the target resource.

`href` An URL pointing to the target resource.

Types

A *type object* is simply a JSON object with the following properties:

body A JSON object where keys are allowed (and required) item names for resource objects and values are JSON Schemas. A JSON Schema is a format for doing data validation on JSON. For now we support the Draft-04 version of the specification (which is the latest supported by the library we use).

To the usual JSON schema types, we add the following ones:

"type": "relationship" This type specifies that the body-item should be a *relationship object*. It supports the following options:

- `arity`, either `to-one`, `to-many` or `auto`.
- `targets`, this option is only valid with `"arity": "to-one"` or `"arity": "to-many"`. It should be the name of a resource type or an array of such names that define which kind of resources can be pointed to by this relationship. If left undefined every type is allowed.
- `pred-type`, this option is only valid with `"arity": "auto"`. It should be a resource type. See more info below.
- `pred-relationship`, this option is only valid with `"arity": "auto"`. See more info below.

"type": "upload" This type specifies that the body-item should be a *blob*. It supports the `acceptable` option that should be an array of content-types that should be accepted.

Automatic relationships

This of relationship description exists because relationships are directed. As such, because sometimes bidirectional relationships are useful, we would want to specify that some relationship is the reverse of another one. To solve that, we introduced the *auto* relationships. This will specify a new relationship that will not be writeable and automatically filled by the Store engine. It will contain as target any resource of the given type that have the current resource as target in the given relationship name. In order to fully specify an `auto` relationship, you need to specify the type of the related object in `pred-type`, as well as `pred-relationship`, the name of the relationship in that object, that should be reversed.

Let's show an example, we will consider two types: users and groups.

```
// user:
 {
     'body': {
         'name': { 'type': 'string' },
         'email': { 'type': 'string', 'format': 'email' },
         'handle': { 'type': 'string' },
         'hash': {'type': 'string'},
         'groups': {
             'type': 'relationship',
             'arity': 'to-many',
             'targets': 'core/group',
         },

         'pinned': {
             'type': 'relationship',
             'arity': 'to-many',
             'targets': 'core/folder'
         }
     }
 }
```

```
// group:
 {
     'body': {
         'name': { 'type': 'string' },
         'handle' : { 'type': 'string' },
         'members': {
             'type': 'relationship',
             'arity': 'auto',
             'pred-type': 'core/user',
             'pred-relationship': 'groups'
         }
     }
 }
```

Now when creating a group you cannot specify it's users, but you can specify the `groups` when creating (or updating) a given user and the system will automagically take care of filling the `members` relationship with the current up-to-date content.

2.3.3 Endpoints

We remind that the API is quite similar to what JSON API proposes. In the following section, type `warrior` is the type defined as:

```
{
    "body": {
        "name": { "type": "string" },
        "honor": { "type": "number"}
        "weapon": {
            "type": "relationship"
            "arity": "to-one",
            "targets": "weapon"
        },
        "kitties": {
            "type": "relationship"
            "arity": "to-many",
            "targets": "cat"
        }
}
```

A warrior has a name and a certain amount of honor. He also possesses a weapon, and can be the (proud) owner of several cats (or no cats).

Resources

Fetching an object

To fetch an object, you must execute a `GET` request on `/api/store/resources/{id}` where `id` is the `ID` of the resource.

Error code:

- `404` if `id` corresponds to no known objects.
- `400` if an error occurred when processing the object (for example, one of the object linked to it doesn't exists anymore in the database).
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *resource object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. Then:

```
>> GET /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e
200
{
   'data':{
      'id':'a0d8959e-f053-4bb3-9acc-cec9f73b524e',
      'href': 'http://tozti/api/store/resources/a0d8959e-f053-4bb3-9acc-
↪cec9f73b524e'
      'type':'warrior',
      'body':{
         'name':'Pierre',
         'honor': 9000
         'weapon':{
            'self':'http://tozti/api/store/resources/a0d8959e-f053-4bb3-9acc-
↪cec9f73b524e/weapon',
            'data':{
               'id':'1bb2ff78-cefb-4ce1-b057-333f5baed577',
               'type':'weapon',
               'href':'http://tozti/api/store/resources/1bb2ff78-cefb-4ce1-b057-
↪333f5baed577'
            }
         },
         'kitties':{
            'self':'http://tozti/api/store/resources/a0d8959e-f053-4bb3-9acc-
↪cec9f73b524e/friend',
            'data':[{
               'id':'6a4d05f1-f04a-4a94-923e-ad52a54456e6',
               'type':'cat',
               'href':'http://tozti/api/store/resources/6a4d05f1-f04a-4a94-923e-
↪ad52a54456e6'
            }]
         }
      },
      'meta':{
         'created':'2018-02-05T23:13:26',
         'last-modified':'2018-02-05T23:13:26'
      }
   }
}
```

Creating an object

To create an object, you must execute a `POST` request on `/api/store/resources` where the body is a JSON object representing the object you want to send. The object must be encapsulated inside a *data* entry.

Error code:

- `404` if one of the object targeted by a relationship doesn't exists
- `400` if an error occurred when processing the object. For example, if the json object which was sended is malformated, or if the body of the request is not JSON.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *resource object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. Then:

```
>> POST /api/store/resources {'data': {'type': 'warrior',
                'body': {
                    'name': Pierre, 'honor': 9000,
                    'weapon': {'data': {'id': <id_weapon>}},
                    'kitties': {'data': [{'id': <kitty_1_id>}]}
                }}}
200
{
   'data':{
      'id':'a0d8959e-f053-4bb3-9acc-cec9f73b524e',
      'type':'warrior',
      'href':'http://tozti/api/store/resources/a0d8959e-f053-4bb3-9acc-
↪cec9f73b524e/',
      'body':{
         'name':'Pierre',
         'honor': 9000
         'weapon':{
            'self':'/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/
↪friend',
            'data':{
               'id':'1bb2ff78-cefb-4ce1-b057-333f5baed577',
               'type':'weapon',
               'href':'/api/store/resources/1bb2ff78-cefb-4ce1-b057-333f5baed577'
            }
         },
         'kitties':{
            'self':'/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/
↪friend',
            'data': [{
               'id':'6a4d05f1-f04a-4a94-923e-ad52a54456e6',
               'type':'cat',
               'href':'/api/store/resources/6a4d05f1-f04a-4a94-923e-ad52a54456e6'
            }]
         }
      },
      'meta':{
         'created':'2018-02-05T23:13:26',
         'last-modified':'2018-02-05T23:13:26'
      }
   }
}
```

Editing an object

To edit an object, you must execute a `PATCH` request on `/api/store/resources/{id}` where `id` is the ID you want to update. The body of the request must be a JSON object representing the change you want to operate on the object. The object must be encapsulated inside a *data* entry. Remark: you don't need to provide every entries.

Error code:

- `404` if `id` corresponds to no known objects.
- `400` if an error occurred when processing the object. For example, if the json object which was sended is malformated, or if the body of the request is not JSON.

- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *resource object* under JSON format representing the object (after changes are applied).

Example: We suppose the object with id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. Then:

```
>> PATCH /api/store/resources {'data': {'type': 'warrior',
                 'attributes': {
                     'name': 'Luc',
                     'weapon': {'data': {'id': <id_weapon_more_powerfull>}},
                 }}}
200
{
   'data':{
      'id':'a0d8959e-f053-4bb3-9acc-cec9f73b524e',
      'type':'warrior',
      'href':'http://tozti/api/store/resources/a0d8959e-f053-4bb3-9acc-
↪cec9f73b524e',
      'body':{
         'name':'Luc',
         'honor': 9000
         'weapon':{
            'self':'/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/
↪friend',
            'data':{
               'id':'<id_weapon_more_powerfull>',
               'type':'weapon',
               'href':'/api/store/resources/<id_weapon_more_powerfull>'
            }
         },
         'kitties':{
            'self':'/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/
↪friend',
            'data': [{
               'id':'6a4d05f1-f04a-4a94-923e-ad52a54456e6',
               'type':'cat',
               'href':'/api/store/resources/6a4d05f1-f04a-4a94-923e-ad52a54456e6'
            }]
         }
      },
      'meta':{
         'created':'2018-02-05T23:13:26',
         'last-modified':'2018-02-05T23:13:26'
      }
   }
}
```

Deleting an object

To delete an object, you must execute a `DELETE` request on `/api/store/resources/{id}` where `id` is the ID you want to update. Remark: you don't need to provide every entries.

Error code:

- `404` if `id` corresponds to no known objects.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back an empty JSON object.

Example: We suppose the object with id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. Then:

```
>> DELETE /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e
200
{}
```

Relationships

In the same way that you can act on resources, you can also act on relationships.

Fetching a relationship

To fetch a relationship, you must execute a `GET` request on `/api/store/resources/{id}/{rel}` where `id` is the ID of the resource possessing the relationship you want to access, and `rel` the name of the relationship.

Error code:

- `404` if `id` corresponds to no known objects or `rel` is an invalid relationship name.
- `400` if an error occurred when processing the object.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *relationship object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. Then:

```
>> GET /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties
↪", "data": [{
            "id": "93b41bf0-73e8-4b37-b2b9-d26d758c2539",
            "type": "cat",
            "href": "/api/store/resources/93b41bf0-73e8-4b37-b2b9-d26d758c2539"
        }, {
            "id": "dff2b520-c3b0-4457-9dfe-cb9972188e48",
            "type": "cat",
            "href": "/api/store/resources/dff2b520-c3b0-4457-9dfe-cb9972188e48"
        }]
    }
}
```

```
>> GET /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/weapon
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/weapon
↪", "data": {
            "id": "34078dd5-516d-42dd-816d-6fbfd82a2da9",
            "type": "weapon",
            "href": "/api/store/resources/34078dd5-516d-42dd-816d-6fbfd82a2da9"
        }
    }
}
```

Updating a relationship

To update a relationship (which is not an automatic relationship), you must execute a `PUT` request on `/api/store/resources/{id}/{rel}` where `id` is the ID of the resource possessing the relationship you want to access, and `rel` the name of the relationship. The content of your request is a JSON object containing:

- for a `to-one` relationship the ID of the new target
- for a `to-many` relationship several IDs representing the new targets

Error code:

- `404` if `id` corresponds to no known objects or `rel` is an invalid relationship name.
- `400` if an error occurred when processing the object.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *relationship object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. We also suppose that its relationship `kitties` possesses two targets having id `<id1>` and `<id2>`. The relationship `weapon` targets `<id_sword>`. Then:

```
>> PUT /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties {'data':␣
↪[{'id': <id3>}]}
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties
↪", "data": [{
            "id": <id3>,
            "type": "cat",
            "href": "/api/store/resources/<id3>"
        }]
    }
}
```

```
>> PUT /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/weapon {'data': {
↪'id': <id_shotgun>}}
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/weapon
↪", "data": [
            "id": <id_shotgun>,
            "type": "weapon",
            "href": "/api/store/resources/<id_shotgun>"
        ]
    }
}
```

Adding new targets to a relationship

To add new targets to a `to-many` relationship, you must execute a `POST` request on `/api/store/resources/{id}/{rel}` where `id` is the ID of the resource possessing the relationship you want to access, and `rel` the name of the relationship. The content of your request is a JSON object containing the ids of the objects you want to add to the relationship.

Error code:

- `404` if `id` corresponds to no known objects or `rel` is an invalid relationship name.
- `403` if the relationship is not a too-many relationship
- `400` if an error occurred when processing the object.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *relationship object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. We also suppose that its relationship `kitties` possesses one targets having id `<id1>`. Then:

```
>> POST /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties {'data
↪': [{'id': <id2>}, {'id': <id3>}]}
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties
↪", "data": [{
            "id": <id1>,
            "type": "cat",
            "href": "/api/store/resources/<id1>"
        }, {
            "id": <id2>,
            "type": "cat",
            "href": "/api/store/resources/<id2>"
        }, {
            "id": <id3>,
            "type": "cat",
            "href": "/api/store/resources/<id3>"
        }]
    }
}
```

Deleting a relationship

To fetch some targets from a `to-many` relationship, you must execute a `DELETE` request on `/api/store/resources/{id}/{rel}` where `id` is the ID of the resource possessing the relationship you want to access, and `rel` the name of the relationship. The content of your request is a JSON object containing the ids of the objects you want to remove from the relationship.

Error code:

- `404` if `id` corresponds to no known objects or `rel` is an invalid relationship name.
- `403` if the relationship is not a too-many relationship
- `400` if an error occurred when processing the object.

- `200` if the request was successful.

Returns: If the request is successful, the server will send back a *relationship object* under JSON format.

Example: Suppose that an object of type `warrior` and id `a0d8959e-f053-4bb3-9acc-cec9f73b524e` exists in the database. We also suppose that its relationship `kitties` possesses three targets having ids `<id1>`, `<id2>` and `<id3>`. Then:

```
>> DELETE /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties {'data
↪': [{'id': <id1>}, {'id': <id3>}]}
200
{
    "data": {
        "self": "/api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/kitties
↪", "data": [{
                "id": <id2>,
                "type": "cat",
                "href": "/api/store/resources/<id2>"
        }]
    }
}
```

```
>> DELETE /api/store/resources/a0d8959e-f053-4bb3-9acc-cec9f73b524e/weapon
403
{
    "errors": [{
        "code": "BAD_RELATIONSHIP",
        "title": "a relationship is invalid",
        "status": "403",
        "detail": "to-one relationships cannot be deleted"
    }]
}
```

Types

Fetching all instances of a given type

To fetch all instances of a given type `<type>`, you must execute a `GET` request on `/api/store/by-type/<type>`.

Error code:

- `404` if the type doesn't exists
- `400` if an error occurred when processing the object.
- `200` if the request was successful.

Returns: If the request is successful, the server will send back a list of linkage objects encapsulated under a *data* entry. Each linkage object points toward a ressources having type `<type>`

Example: To fetch every `warrior` present inside our `store`, you can proceed as following:

```
>> GET /api/store/by-type/warrior
200
{
    "data": [
    {
        "id": "60f1677b-2bbb-4fd9-9a7a-3a20dbf7b5af",
        "type": "core/user",
        "href": "/api/store/resources/60f1677b-2bbb-4fd9-9a7a-3a20dbf7b5af"
    }, {
        "id": "605ab4bc-172b-416e-8a13-186cf3cd1e2e",
        "type": "core/user",
        "href": "/api/store/resources/605ab4bc-172b-416e-8a13-186cf3cd1e2e"
    }]
}
```

Remark: Most of the time, type names are under this form: `<ext-name>/<type-name` where `<ext-name>` is the name of the extension defining the type `<type-name>`. To fetch of instances of this type, send a `GET` request on `/api/store/by-type/<ext-name>/<type-name>`.

2.4 Developing Extensions

2.4.1 Getting Started

Our first extension

Let's see how to create a simple extension to *tozti*. Everything defined by an extension lives inside the same folder, whose name is the name of the extension.

Suppose we call it `extension-name`. Browse to the `extensions/` folder and proceed to create a folder `extension-name`. The only requirement for *tozti* to recognize an extension is for this extension to provide a file `server.py` declaring a dictionnary `MANIFEST`. Thus a minimal definition would be like so:

```
MANIFEST = {
  'name': 'extension-name',
}
```

Well done, you've just created your first extension!

Defining an API endpoint

The previous extension currently does nothing. We will now see how to add new API endpoints to the application.

At the moment our `MANIFEST` is empty. To declare new routes, we must import some modules:

```
from tozti.utils import RouterDef
from aiohttp import web
import logbook
```

- `RouterDef` allows us to define a new router and therefore new request handlers.
- `web` from `aiohttp` enables us to send back to the user simple responses.
- `logger` is a simple utility to pretty print information in the server logs.

We define a logger, which will enable us to output useful information to the console:

```
logger = logbook.Logger("tozti-routing")
```

Then, we create an empty router:

```
router = RouterDef()
```

And we add one endpoint to it. We call it `hello_world`, and make it accessible from the URL `<tozti>/api/extension-name/hello_world`:

```
hello_world = router.add_route('/hello_world')
```

Finally, we define how our extension should behave when this endpoint is requested. In this example, we respond to `GET` requests on this endpoint with some dummy text:

```
@hello_world.get
async def hello_world_get(req):
    logger.info("hello world")
    return web.Response(text='Hello world!')
```

Similar decorators are available for the usual HTTP methods: `@hello_world.post`, etc.

Unfortunately, for now *tozti* still isn't aware of this new request handler we just defined. This is where `MANIFEST` comes into use: We simply add the router in the `MANIFEST` dict under the key `router`:

```
MANIFEST = {
    'name': 'extension-name',
    'router': router,
}
```

In fact, `MANIFEST` is where we declare anything that *tozti* should be made aware of.

And now, if you launch the server again, and visit the URL `<tozti>/api/extension-name/hello_world`, your web browser should display a blank web page with the text *"Hello world!"*. If you look in the server logs, some `hello world` must have appeared.

Providing custom javascript to the tozti application

If the previous paragraph showed how to serve content on specific URLs, this is *not* how we modify the behavior of the *tozti* application. *tozti* is a single-page app built with the framework **Vue.js**. Therefore if you want to be able to interact with the application and define new interactions, you need to be able to serve custom *javascript* code to the client.

As a convention, all static assets must be put inside a folder `dist` inside your extension folder. Let's create a file called `index.js` inside `extension-name/dist/`:

```
tozti.addRoutes([
  { path: '/extension', component: {
      template: '<div>my extensions</div>'
    }
  }
])
```

As you might have guessed, we need to inform *tozti* of the existence of this file, inside `MANIFEST`:

```
MANIFEST = {
  # ..
  'includes': ['index.js']
}
```

Once again, start the server and visit the URL `<tozti>/extension`. You should see some new content inside the usual layout.

As stated below, adding CSS files in this `includes` list in exactly the same fashion allows the inclusion of custom CSS to *tozti*.

Quick note on file structure

Most extensions do not serve directly their javascript files to *tozti*. They often split their code in separate files, and use some build process to obtain a single file `build.js` out of their code. This is the file that they send to the client. We will not describe here how to setup such a build process, as it would end up very much opinionated, and still would have to differ between extensions. However it is very much recommended to proceed in such a way, and the sample extensions available on our github page provide some insight as to how things can be organised.

Going further with `MANIFEST`

Here are a complete list of keys that `MANIFEST` can possess:

`name` The name of the extension, in lower-case and with dashes instead of spaces.

`router` This is used to declare new API endpoints. It should be an instance of *`tozti.utils.RouterDef`*. More precisely it must have an `add_prefix()` method and it will be passed to `aiohttp.web.UrlDispatcher.add_routes()`. Every route declared will be prefixed by `/api/<extension-name>`.

`includes` A list of css or js filenames that must be included in the main `index.html`. Usually you will put there your `main.js` which contains the code to register or patch components.

`dependencies` A list of names of extensions that must be loaded before this extension in order for it to be working as intended.

For more advanced user, you can also add signals for the *aiohttp.web* in the `MANIFEST`. Please see aiohttp server documentation to learn more about signals.

`_god_mode` Beware, this can be dangerous if used incorrectly! This should be a function taking as argument the main `aiohttp.web.Application` object. You can use it to register custom middlewares or do otherwise weird stuff.

`on_response_prepare` This should be a function. It is a hook for changing HTTP headers for streamed responses and WebSockets.

`on_startup` This should be a function. Will be called during the startup of the application. Usefull to launch background services for exemple.

`on_cleanup` This should be a function. Will be called on application cleanup. You can use it to close connections to the database for exemple.

`on_shutdown` This should be a function. Will be closed on application shutdown.

Having a more complex server

Sometimes you can find that putting the whole server part inside `server.py` is a bit too restrictive. As your extension grow you'll probably want to refactor it in several files. Tozti provide a way to do so. Instead of creating a `server.py` file, you could create a `server/` folder, and inside it write a file `__init__.py` defining (at least) the `MANIFEST` structure.

2.4.2 Using Tozti's JS api

Defining routes on the client side

If you read *Getting Started* you learned how to define new API endpoints. But you might want that your extension also provide some endpoints on the client, to display a special page for example.

You can take a look at how the extension *vue-counter* of the sample-extensions repository uses this mechanism to define a counter reachable on `<tozti>/counter`.

Tozti's extensions are using vue, so it is natural that we use `vue-router` in order to define new routes.

Imagine you want to define a new 'page' displaying a component called `Calendar` that can be accessed on `<tozti>/mycalendar`. Then, you must add the following lines in your `index.js`:

```
tozti.addRoutes([
  { path: '/mycalendar', component: Calendar }
])
```

Adding items in the menu bar

An exemple can be found in the extension *add-menu-item* that can be found in the repository sample-extensions.

Every extensions can add items in the sidebar. We will focus on what we call *menu items*: items that are attached to tozti as a whole, not to a workspace.

The corresponding method allowing to do that is called `tozti.addMenuItem`. Here are following examples of usage:

- Adding an item with the text 'item' associated with the route *target/*:

```
tozti.addMenuItem('item', 'target/')
```

- It is possible to specify which icon to use with the item inside the sidebar. For this, you simply have to give additional properties as a third parameter. The *icon* field should be a CSS class from the Nucleo App Free Icon set. (This might & most surely will change later on).

```
tozti.addMenuItem('item', 'target/', {'icon': 'nc-home-52'})
```

2.4.3 Communication with the storage

More often than not, the purpose of an extension is to define new types of resources, and provide new interactions with them from the tozti interface. For this purpose, tozti provides an easy way to define new types on the server-side, and a generic client-side API to query the store from the frontend.

Defining new types of resources

For validation purposes, when you want to create a new type of resource in the store, you need to specify how such a resource is structured. This is done via **schemas**, similar to the JSONSchema specification.

New type schemas can be defined from an extension by providing a new entry called `types` in the `MANIFEST` of the extension. This entry should be a dictionary, whose keys are the names of the new types, and values are valid schemas.

Let's show how this works with a simple example, with our dummy extension. We would like to define a new resource type called `entity`, with attributes `name` and `age`. For the sake of it, let it also have a `to-many` relationship named `friends`.

Therefore, our MANIFEST (defined in server.py) should look like:

```
MANIFEST = {
  'types': {
    'entity': {
      'body': {
        'name': { 'type': 'string' },
        'age': { 'type': 'integer' },
        'friends': {
          'type': 'relationship'
          'arity': 'to-many',
          'targets': 'dummy-extension/entity',
        }
      }
    }
  }
}
```

Note that in the rest of this documentation, extension-defined type names will be prefixed by the name of the folder containing the extensions. Assuming our extension lives in a folder called dummy-extension in the extensions folder of our main tozti instance, the newly defined type will now be referred as dummy-extensions/entity. This also applies to the core types. For example, we provide by default types core/user, and core/group. That is why in the relationship friends of the type we just defined, we refer to dummy-extensions/entity.

Further information on type schemas and how the storage uses them can be found in the documentation specific to the storage. (TODO: add link).

Accessing the store from the JS API

Resources

Now that we have registered a new type for resources in the store, we would like to have the ability to interact with such resources. tozti provides a generic API for this end, under the tozti.store namespace.

Getting a resource

When you have the *uuid* of a resource, you can get its data from the store by using the get method:

```
tozti.store.get(uuid)
```

This method returns a javascript **promise**, that resolves to the resource object, or rejects to the HTTP response object.

For exemple, assuming the variable uuid contains the uuid of a resource of type dummy-extension/entity, we can print the name of said entity by doing:

```
tozti.store
  .get(uuid)
  .then(resource => {
    console.log(resource.body.name)
  })
  .catch(response => {
    console.error('An error occured while fetching the resource.')
  })
```

Danger: You should never write to the resource objects given by the `tozti.store` methods, ever. Think of them as readonly objects. If you want to update a resource, see `tozti.store.update` introduced below.

Creating a new resource

To populate the store from the client-side, you have the ability to create new resources and send them to the server store. First define a new resource object:

```
let resource = {
  type: 'dummy-extension/entity',
  body: {
    name: 'Some Entity',
    age: 15,
    friends: {
     data: []
    }
  }
}
```

The only required field is the `type` field, for the storage to know what you are trying to create. Note that the associated `type` schema may itself require you to specify other fields.

Then, you can create the resource and send it to the store with the `create` method:

```
tozti.store.create(resource)
```

This method also returns a javascript **promise**, that resolves to the full store resource object, or rejects to the HTTP response object. The resolved resource is a fully defined store resource, so it contains `href`– a URL to itself, a `meta` field with meta information, and a `body` filled with the actual properties of the resource. It also has an `id` field, which contains the UUID of the resource inside the remote store.

```
tozti.store
  .create(resource)
  .then(res => {
    console.log('The resource was created.')
    console.log(res.id)
  })
  .catch(response => {
    console.error('An error occured while creating the resource.')
  })
```

Updating a resource

Another usual operation is updating an existing resource. This is done via the `update` method.

First, you need to define a resource object containing only the items that you want to see updated, and at the very least an `id`.

```
let changes = {
  id: 'some-resource-id',
  body: {
    name: 'A new name for the entity',
  }
}
```

Then, using the `update` method will try to apply the changes to the server:

```
tozti.store.update(changes)
```

Again, this method returns a javascript **promise**, that resolves to the full store resource object, with the applied changes.

```
tozti.store
  .update(changes)
  .then(resource => {
    console.log(resource.body.name)
    // expected output: A new name for the entity
  })
  .catch(response => {
    console.error('An error occured while updating the resource.')
  })
```

Deleting a resource

Finaly, to delete a resource from the store, one can use the `delete` method. This method takes a resource object as a parameter, whose only needed field is `id`. (The fact that it takes a resource object is for convenience only).

Assuming, `uuid` contains some entity id, and `resource` contains a complete resource object coming from the store, it can be used like this:

```
tozti.store.delete({ id: uuid })
tozti.store.delete(resource)
```

As expected, this method also returns a **promise**, which resolves to an empty object when the deletion was successful, or to the HTTP response in the eventuality of an error.

Relationships

If some resource has a relationship, then in the resource object returned from the store, the associated relationship field only contains a *linkage*, or an array of *linkages*.

(Recall that a *linkage* is simply an object referring to a resource, containing fields `type` and `id`, plus additional data)

tozti provides helper functions for fetching the entire data of a relationship, and updating it, in the `tozti.store.rels` namespace.

Getting the resources of a relationship

To get all the resources pointed by a relationship, use the `rels.fetch` method. It takes as a parameter a relationship object coming from some resource object returned by the store, and returns a Promise.

This promise either resolves to a single resource object when the relationship is `to-one`, or to an array of resource objects when the relationship is `to-many`.

This promise is rejected if any of the resources contained in the relationship cannot be accessed from the server. For this reason, and for a better UX experience, it is preferred to not use `fetch` but instead defer the responsability of loading contained resources to individual components, that can display errors more intuitively. (See: part on using the store from Vue components, further down on the same page)

Assume that we have a resource `resource` of type `dummy-extension/entity`, then we can get all resources contained in the `friends` relationship by doing:

```
tozti.store.rels
  .fetch(resource.body.friends)
  .then(friends => {
    // log the name of every friend in the relationship
    friends.forEach(friend => {
      console.log(friend.body.name)
    })
  })
  .catch(response => {
    console.error('An error occured while fetching some entities of the relationship')
  })
```

Appending resources to a *to-many* relationship

`rels.add` allows you to add some resource to a relationship. Its first parameter is a relationship object. All the other arguments will be interpreted as linkages to be added to the relationship. It returns a promise resolving to the new relationship object. Note that the original relationship object is actually mutated to correspond to the new relationship data. The linkages provided only need to define an `id` field.

Assuming we have two resources `pomme`, `poire`, `abricot` of type `dummy-extensions`, adding `poire` and `abricot` to the relationship `friends` of resource `pomme` is done like this:

```
tozti.store.rels
  .add(pomme.body.friends, { id: poire.id }, abricot)
```

If some linkages already exist inside the relationship, they will not be added twice but the promise will still resolve correctly to the relationship object.

Removing resources from a *to-many* relationship

`rels.delete` does the exact opposite of `rels.add`: it allows you to remove some resources from a relationship. It takes a relationship object as first parameter, and any other argument will be interpreted as a linkage. It returns a promise resolving to the new relationship object.

Again, the original relationship object is actually mutated to correspond to the new relationship data. Linkages provided only need to hold an `id` field.

Using the same exemple as before, we now want to remove `poire` and `abricot` from the relationship `friends` of resource `pomme`:

```
tozti.store.rels
  .delete(pomme.body.friends, poire, { id: abricot.id })
```

If some linkages do not exist inside the relationship, they will simply be ignored, and the promise will still resolve correctly to the relationship object.

Updating a relationship

Unimplemented yet. This will be added soon.

Accessing data from Vue components

A nice feature that was purposefully ignored earlier, is the fact that the JS API keeps a local version of the store. What this means is that when someone uses `tozti.store.get` with the id of a resource that was already fetched somewhere else, the promise will immediately resolve to **the same cached resource object**.

Likewise, every operation sent to the remote storage will be applied to the cached version of the resource, if it exists. For example, `tozti.store.update` will locally mutate the cached target resource to sync with the server version.

This is especially useful in that it enables reactivity without even having to think about it. Simply update a resource and the changes will be seen everywhere the resource is being used, on the frontend, without actually having to request the data from the server again.

Below, we will look at how one can actually use the store API to fetch data inside Vue components.

Resource components

Usually, it is good to use specific components to display a single resource. Be it inside a list of items, or a single page displaying information about the resource, working with individual components that care about a single resource at the time is easier to reason about and compose into more involved components.

Let's define a component called `EntityView` that will display information about one resource. For the component to know which resource it is being associated with, we need to give it an `id`, through props. As soon as the component is being used (i.e mounted), we want the component to fetch the necessary data from the store. Finally, as long as the data request is being processed, we simply cannot show any data, so we need to make sure that the loading is explicit inside the component.

This would give something similar in the vein of:

Caution: This is given as an exemple, but we would prefer you using `resourceMixin`, introduced right after.

```
<template>
  <div>
    <p v-if="resource">
      Name: {{ resource.body.name }} <b>
      Age:  {{ resource.body.name }}
    </p>
    <p v-else>
      The resource is being loaded.
    </p>
  </div>
</template>
<script>
  export default {
    props: {
      id: {
        type: String
      }
    },

    data() {
      return {
        resource: null
      }
    },
    beforeMounted() {
      tozti.store.get(this.id)
        .then(resource => {
          this.resource = resource
        })
    }
  }
</script>
```

Then the component can be used with `<entity-view :id="some-resource-id"></entity-view>`.

This should work properly: we query the data when the component is mounted, and conditionally display the content once the resource has been returned. However, making this work reliably is more involved, since components can be reused and repurposed freely by Vue.

To make it easier for developers to define this kind of components, we provide a default mixin:

```
<template>
  <div>
    <p v-if="!loading">
      Name: {{ resource.body.name }} <b>
      Age:  {{ resource.body.name }}
    </p>
    <p v-else>
      The resource is being loaded.
    </p>
  </div>
</template>
<script>
  import { resourceMixin } from 'tozti'

  export default {
    mixins: [ resourceMixin ]
  }
</script>
```

This mixin defines two data fields:

- `loading`, a boolean that indicates whether a data request is currently being processed.
- `resource`, initially set to `null`, that will contain the resource once it has been acquired.

Use this mixin as soon as it may be suitable!

Displaying relationships

Using relationships inside Vue components is quite similar. Here we will describe the common patterns related to **to-many** and **to-one** relationships.

to-many relationships

When one wants to display the associated resources of some relationship, the preferred pattern is to simply display a list of components, that will each be responsible for displaying a single resource of the relationship. This allows for fine-grained error handling, when one of the resources no longer exists. In such a situation, the associated component can simply display an error message, without affecting the other components.

To react to relationship changes, you need to add the relationship `data` array inside the data of your main component, that will contain every linkage included in the relationship.

In our example, we assume that we are defining a global component, that displays a given `dummy-extension/entity`'s friends:

```
<template>
  <div v-if="!loading">
    <entity-view v-for="friend in friends" :id="friend.id" :key="friend.id">
    </entity-view>
  </div>
</template>
<script>
  import { resourceMixin } from 'tozti'

  export default {
    mixins: [ resourceMixin ],

    computed: {
      friends() {
        // this.friends will contain an array of linkages
        // and will be computed when the main resource is finally ready
        return this.resource.body.friends.data
      }
    }

  }

</script>
```

In this exemple, we defer the responsability of loading individual resources to the `EntityView` component defined earlier. The relationship data array will be watched by Vue, therefore when the relationship is updated somewhere inside the client, the interface should be updated without needing further work.

2.5 API Reference

2.5.1 `tozti.utils`

exception `tozti.utils.`**`APIError`**(*template=None*, *status=None*, ***kwargs*)

Base class for API errors.

`to_response`()

Create an *aiohttp.web.Response* signifiying the error.

class `tozti.utils.`**`ExtendedJSONEncoder`**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

JSON encoder handling *datetime.datetime* and *uuid.UUID*.

class `tozti.utils.`**`RouteDef`**(*path*, *name=None*)

Definition of a route.

The method `get()`, `post()`, `put()`, etc can be used as decorators to specify the handler for the given HTTP method.

any (*handler*)
Decorator used to specify handler for every method.

delete (*handler*)
Decorator used to specify `DELETE` method handler.

get (*handler*)
Decorator used to specify `GET` method handler.

head (*handler*)
Decorator used to specify `HEAD` method handler.

options (*handler*)
Decorator used to specify `OPTIONS` method handler.

patch (*handler*)
Decorator used to specify `PATCH` method handler.

post (*handler*)
Decorator used to specify `GET` method handler.

put (*handler*)
Decorator used to specify `PUT` method handler.

register (*app*)
Add all our routes to the given *aiohttp.web.Application*.

route (**meth*)
Decorator (with arguments) used to specify HTTP handler.

class tozti.utils.**RouterDef**
Handle route definitions.

This object can be used as argument to `aiohttp.web.UrlDispatcher.add_routes()`.

Sample usage:

```
router = RouterDef()
route = router.add_route('/foo')

@route.get
def handle_get(req):
    return ...
```

See aiohttp for more informations on resources and routing.

add_prefix (*prefix*)
Prefix every contained route.

add_route (*path*, *name=None*)
Add and return a route with given path to the router.

tozti.utils.**json_response** (*data*, ***kwargs*)
Wrapper for *aiohttp.web.json_response* with extended JSON encoder.

tozti.utils.**validate** (*inst*, *schema*)
Validate data against a JsonSchema.

2.5.2 `tozti.store`

2.5.3 `tozti.app`

CHAPTER 3

Project internals

For the internal organisation, workflows and anything related to the *software project* course, see meta/index. If you want to write tests, a small documentation is available at meta/tests.

Python Module Index

Index