User Guide for Request Method using request templates

Status: Closed Beta

Platform 2.3 introduces a new way of sending HTTP requests using Request Method. App developers can now define request templates statically in the app's configuration and invoke those named templates to send an HTTP request from the app.

This document covers how to use this new feature and how to migrate any existing Request Method code to this new model.

Table of content

What's new

- HTTP request configurations will now need to be added in a new configuration file: `config/requests.json`.
- App manifest gets a new property called `"requests"` in each product to explicitly describe which templates defined in the configuration file can be used in the app code.
- The `"whitelisted-domains"` property is not needed anymore in the app manifest as the request template explicitly declares the host.
- A new runtime API called `client.request.invokeTemplate()` for frontend and `$request.invokeTemplate()` for serverless is introduced to work with request templates.
- Client JS `src` in frontend HTML code can be replaced with `{{{appclient}}}`.

How to send an HTTP request

Step 1: Configure request templates

The first step in sending an HTTP request is to create a configuration in `config/requests.json`. This file exports an object where each key is the name of the request template and its value is an object that describes the request template configuration (see “Request schema” below).

Each request template configuration consists of two properties: `"schema"` and `"options"`.

In the code below, `"getTickets"` is the name of the template, while its schema and options are described in nested properties:

./config/requests.json:

```
{
  "getTickets": {
    "schema": {
      // ... request schema
    },
    "options": {
      // ... request options
    }
  }
}
```

Request schema

The request schema can contain the following fields:

Name	Type	Required	Default
`"method"`	`String`	**Yes**	-
`"protocol"`	`String`	No	`"https"`
`"host"`	`String`	**Yes**	-
`"path"`	`String`	No	`"/"`
`"headers"`	`Object<String, String>`	No	`{ }`
`"query"`	`Object<String, String|Number>`	No	`{ }`

These fields directly correspond to an HTTP request's properties. Each request schema must specify `"method"` and `"host"`. Let's look at the fields in detail:

1. **method:** The HTTP request method. Can be either `"GET"`, `"POST"`, `"PUT"`, `"DELETE"`, or `"PATCH"`.
2. **protocol:** The only supported protocol in production is `"https"`. `"http"` is supported only in local testing, so you cannot pack and upload an app that uses `"http"` in a request template's protocol. It's optional to add since the platform allows you to connect only to a secure endpoint in production.
3. **host:** The host server to send the request to, aka, the domain name. Must be an FQDN and not an IP address. Don't prepend the protocol and don't append a trailing slash. E.g., `"example.freshdesk.com"`.
4. **path:** The path to the requested resource on the host. If not specified, this defaults to the root path. Avoid adding query parameters here as they can

go in the `"query"` field. Begin with a trailing slash. E.g., `"/api/v2/tickets"`.

5. **headers:** Request headers are specified as an object in key-value pairs. All values must be strings.
6. **query:** URL query parameters are added as an object in key-value pairs.

[Note]: Request body is passed at runtime and not added to the template.

Template substitutions and variables

Out of these fields , **host** , **path** , **headers** and **query** allow using template substitutions to add variables. The general syntax looks like this: `"<%= variableName %>"`.

iParam values

iParams values can be accessed through variable names in the form of `iparam.variableName` where `variableName` is the name of the iParam. For example, to use an iparam called `subdomain` in the host field, you would write something like: `"<%=iparam.subdomain %>.freshdesk.com"`.

Template substitutions also allow you to use the `encode()` function to encode an API Key. A common usage of this function is paired with a secure iparam to pass an encoded API key in the `"Authorization"` header, like this:

```
"Authorization": "Bearer <%= encode(iparam.api_key) + ':x' %>"
```

Pass runtime data using context

In addition to iparams, request schemas can also allow runtime code to pass certain data to the requests dynamically. These variables are available in the form of `context.variableName`.

These are useful for passing data like pagination query parameters, allowing the same template to be used to fetch all pages of a paginated endpoint:

```
"query": {
  "page": "<%= context.page %>",
  "per_page": "20"
}
```

[Note]: See the “Step 3: Runtime API” section to know how to pass context data when triggering a request.

Variable limitations

Given security concerns, not all kinds of variables can be used everywhere. The following table shows what variables can be used for which fields in the request schema:

Property	Template substitution	Variables supported
`"method"`	No	None
`"protocol"`	No	None
`"host"`	Yes	Only non-secure iparams
`"path"`	Yes	Non-secure iparams, Context
`"headers"`	Yes, only in values	Secure & non-secure iparams, OAuth `access_token`, Context
`"query"`	Yes	Non-secure iparams, Context

Request options

The `"options"` property of a request template takes the following values, all of which are optional:

Name	Type	Valid values	Default	Description
`"maxAttempts"`	`Number`	`1` - `5`	`1`	The maximum number of times a request will be resent if a network or 429/5xx HTTP error occurs.
`"retryDelay"`	`Number`	`1` - `1500`	`1000`	Delay between retry requests, specified in milliseconds.
`"isOAuth"`	`Boolean`	`true`, `false`	`false`	Send an OAuth request. Use the `access_token` variable to access the token

Step 2: Declare templates in app manifest

Once the request templates are in place, the next step is to declare which of those templates should be made available to a product's runtime code.

This is done by adding a `"requests"` property under each product in `manifest.json`. The `"requests"` property is an object whose keys can be one of the request template names as declared in `./config/requests.json`, and their values are the request options object. Any request options specified in manifest.json will override the ones specified in `./config/requests.json`.

`manifest.json`:

```
{
  "platform-version": "2.3",
  "product": {
    "freshdesk": {
      "events": {
        "onTicketCreate": {
          "handler": "onTicketCreateHandler"
        }
      },
      "requests": {
        "createTicket": {},
        "getTickets": {}
        }
      }
    }
  },
  "engines": {
    "node": "14.19.1",
    "fdk": "9.0.0"
  }
}
```

The example above declares two request templates, `"createTicket"` and `"getTickets"`, for Freshdesk, both of which must be defined in

`./config/requests.json`.

This explicit declaration allows the same templates to be reused across different products in an omni-app while ensuring no unwanted HTTP requests are triggered by app code. It also allows overriding the request options, which can be useful when importing request configurations from external sources.

Step 3: Runtime API

Once all the configuration is in place, the final step is to invoke the request templates from the app code, frontend or serverless, and pass any data that the template wants.

To do this, call the **`client.request.invokeTemplate()` method from frontend code**, and **`$request.invokeTemplate()` from serverless code**.

Frontend code:

```
await client.request.invokeTemplate(requestName, {
  context: {}, // optional
  body: JSON.stringify(body),
});
```

Serverless code:

```
await $request.invokeTemplate(requestName, {
  context: {}, // optional
  body: JSON.stringify(body),
});
```

The `.invokeTemplate()` method takes two arguments:

1. The first argument is the request template name that is to be invoked. e.g., `"createTickets"` .
2. The second argument is an object with these properties:
   i. **`"context"`:** An object containing the `context` variables that the template needs.
   ii. **`"body"`:** Request body, if needed, as a string.

It returns a `Promise` that resolves to the response data or an error, similar to calling the erstwhile request method runtime API.

[Note]: It is not possible to change request options at runtime. Request options must be explicitly specified in `config/request.json`. In the future, overriding request options might be allowed in `manifest.json`.

Example

`./config/requests.json`:

```
{
  "getTickets": {
    "schema": {
      "method": "GET",
      "host": "<%= iparam.domain %>.freshdesk.com",
      "path": "/api/v2/tickets",
      "headers": {
        "Authorization": "Bearer <%= iparam.apikey %>",
        "Content-Type": "application/json"
      },
      "query": {
        "page": "<%= context.page %>",
        "per_page": "20"
      }
    },
    "options": {
        "retryDelay": 1000,
        "per_page": 10
    }
  },
  "sendToExternalAPI": {
    "schema": {
      "method": "POST",
      "host": "<%= iparam.ext_domain %>.example.com",
      "path": "/api/",
      "headers": {
        "Authorization": "Bearer <%= iparam.ext_apikey %>",
        "Content-Type": "application/json"
      }
    },
    "options": {
      "timeout": 5
    }
  }
}
```

`manifest.json`:

```
{
  "platform-version": "2.3",
  "product": {
    "freshdesk": {
      "location": {
        "ticket_sidebar": {
          "url": "index.html",
          "icon": "styles/images/icon.svg"
        }
      },
      "events": {
        "onTicketCreate": {
          "handler": "onTicketCreateHandler"
        }
      },
      "requests": {
        "getTickets": {},
        "sendToExternalAPI": {}
      }
    }
  },
  "engines": {
    "node": "14.19.1",
    "fdk": "9.0.0"
  }
}
```

`./server/server.js`:

```
exports = {
  async onTicketCreateHandler(args) {
    // Send new ticket details to the external API
    await $request.invokeTemplate("sendToExternalAPI", {
      body: JSON.stringify(args.data),
    });
  },
```

```
};
```

`./app/scripts/app.js`:

```
// ------
// Setup
// ------
document.onreadystatechange = function () {
  if (document.readyState === "interactive") {
    app
      .initialized()
      .then(function (client) {
        client.events.on("app.activated", function () {
          onAppActivate(client);
        });
      })
      .catch(handleErr);
  }
};

// <-- snip -->

// ------
// Logic
// ------
async function onAppActivate(client) {
  // Get tickets for page 2
  await client.request.invokeTemplate("getTickets", {
    context: {
      page: 2,
    },
  });
}
```

Migrating existing request method code

Since this is a big shift from existing request to request methods using request templates, this section explains how to identify the parameters for migrating to the new request methods using request templates.

Step 1: Extract configuration

When migrating existing `client.request.*()` calls in your front-end app code or `$request.*()` calls in your serverless app code, you need to first identify the requests which are used across the app.

Consider the following function:

```
async function getContacts(client) {
  const url = "https://<%= iparam.subdomain %>.freshdesk.com" +
  "/api/v2/contacts?page=2";
  let options = {
    headers: {
      Authorization: `Basic <%= encode(iparam.api_key) %>`,
      "Content-Type": "application/json",
    },
    maxAttempts: 2,
    retryDelay: 100
  };
  let response = await client.request.get(url, options);
  // logic with the response ....
}
```

In order to convert this function to use the new request method templates, let us first give this new request template a name. We'll name it `"getContacts"`.

Then, we need to extract the details for the configuration we need to add in `./config/requests.json`.

The following go in the `"schema"` section of the request template:

- `"protocol"`: `"https"`
- `"method"`: `"GET"`
- `"host"`: `"<%= iparam.subdomain %>.freshdesk.com"`
- `"path"`: `"/api/v2/contacts"`
- `"headers"`: The headers in this example can be copied over, with the only being making it JSON-compatible (quotes around property names):

```
{
  "Authorization": "Basic <%= encode(iparam.api_key) %>",
  "Content-Type": "application/json"
}
```

- `"query"`: Given that the `"page"` query can be parameterised, we can use `context` here to pass the page number at runtime, instead of hard-coding the page value:

```
{
  "page": "<%= context.page %>"
}
```

Then, extracting the request options, we get the following properties that go in the `"options"` property of the request template:

- `"maxAttempts"`: `2`
- `"retryDelay"`: `100`

Step 2: Create request template

With this information, our `./config/requests.json` file would look like this:

New `./config/requests.json`:

```
{
  "getContacts": {
    "schema": {
      "method": "GET",
      "protocol": "https",
      "host": "<%= iparam.subdomain %>.freshdesk.com",
      "path": "/api/v2/contacts",
      "headers": {
        "Authorization": "Basic <%= encode(iparam.api_key) %>",
        "Content-Type": "application/json"
      },
      "query": {
        "page": "<%= context.page %>"
      }
    },
    "options": {
      "maxAttempts": 2,
      "retryDelay": 100
    }
  }
}
```

Step 3: Update manifest.json

Next, we need to declare the newly created request template in the `manifest.json`:

Updated `manifest.json`:

```
{
  "platform-version": "2.3",
  "product": {
    "freshdesk": {
      "location": {
        "ticket_sidebar": {
          "url": "ticket.html",
          "icon": "styles/images/icon.svg"
        }
      },
      "events": {
        "onTicketCreate": {
          "handler": "onTicketCreateHandler"
        }
      },
      "requests": {
        "getContacts": {}
      }
    }
  },
  "engines": {
    "node": "14.19.1",
    "fdk": "9.0.0"
  }
}
```

Notice that:

1. We have changed the `"platform-version"` property's value to `"2.3"`, while the `engines.fdk` property's value now points to `9.0.0`. Request templates are supported in platform version `>= 2.3` and FDK `>= 9.0.0`

2. We have added a new `"requests"` property under `"freshdesk"`, with the template name as the key of an empty object.
3. We have removed the `"whitelisted-domains"` property as it is not needed in platform 2.3.

Step 4: Update app code

Update client JS resource in frontend HTML

We need to replace existing client JS script `src` in frontend HTML files. In this example, this would be in `ticket.html`:

From:

```
<script
  src="https://static.freshdev.io/fdk/2.0/assets/fresh_client.js">
</script>
```

To:

```
<script src="{{{appclient}}}"></script>
```

Replace runtime API usage

We can now rewrite the `async function getContacts()` that we had started with to use the new `client.request.invokeTemplate()` method:

```
async function getContacts(client, page = 1) {
  let res = await client.request.invokeTemplate("getContacts", {
    context: {
      page,
    },
  });
  // logic with the response ....
}
```

So, any function calling the updated `getContacts()` function can pass in the page number that we pass along to the template using `context`.

Set up both FDK 8 and FDK 9

If you are a developer who needs to maintain apps on both platform version 2.2 and 2.3 for a while, you can keep both FDK 8 and FDK 9 installed and set up on your machine.

The main idea is to have one FDK major version installed globally, while another FDK version is setup locally and aliased to some handy command. In this case, you would want to keep FDK 9 installed globally (so that it can receive updates easily), and setup the latest FDK 8 locally and alias it.

Watch this short video to see the process:

https://www.youtube.com/watch?v=xupb2Yi2N_A

Step 0: Cleanup

Remove FDK's user-level data directory:

```
rm -rf ~/.fdk
```

Uninstall globally installed FDK, just in case:

```
npm rm fdk -g
```

Step 1: Install FDK 9 globally

```
npm install https://cdn.freshdev.io/assets/cli/fdk.tgz -g
```

Once installation succeeds, test the FDK version, you should see something like:

```
$ fdk version
Installed: 9.0.0
Already Up to Date..!
```

Step 2: Setup FDK 8 locally

Instead of installing the package using the v8 tarball URL, download the tarball and extract it instead. I prefer to keep these user-local binaries in `~/bin/`.

```
mkdir -p ~/bin
cd ~/bin
```

Get the tarball:

```
curl https://dl.freshdev.io/cli/fdk.tgz > fdk8.tgz
```

Extract it and rename the extracted directory to something handy, like `fdk8`:

```
tar -xzf fdk8.tgz
mv package fdk8
```

Change to the fdk8 directory and install package dependencies locally:

```
cd fdk8
npm install
```

The entry point for the fdk package is `index.js`, so you can do:

```
node index.js version
```

This should show you:

```
$ node index.js version
Installed: 8.6.6
Already Up to Date..!
```

Step 3: Alias fdk8

Set up a quick alias using:

```
alias fdk8="node ~/bin/fdk8/index.js"
```

This will only be available for the current session of the shell.

If you want to persist this change, add the command above as a new line to your `~/.bashrc`, or `~/.zshrc` or whichever file your shell loads at startup, except provide absolute path to the `index.js` file, for example:

```
alias fdk8="node /Users/kadas/bin/fdk8/index.js"
```

Step 4: Try them out

You should now be able to use the `fdk` command for FDK 9 and the `fdk8` command for FDK 8.