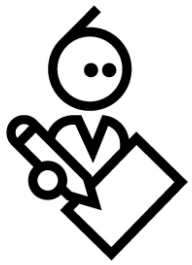
Writing a uNabto HTML client application

NABTO/001/TEN/024



Contents

1 Abstract

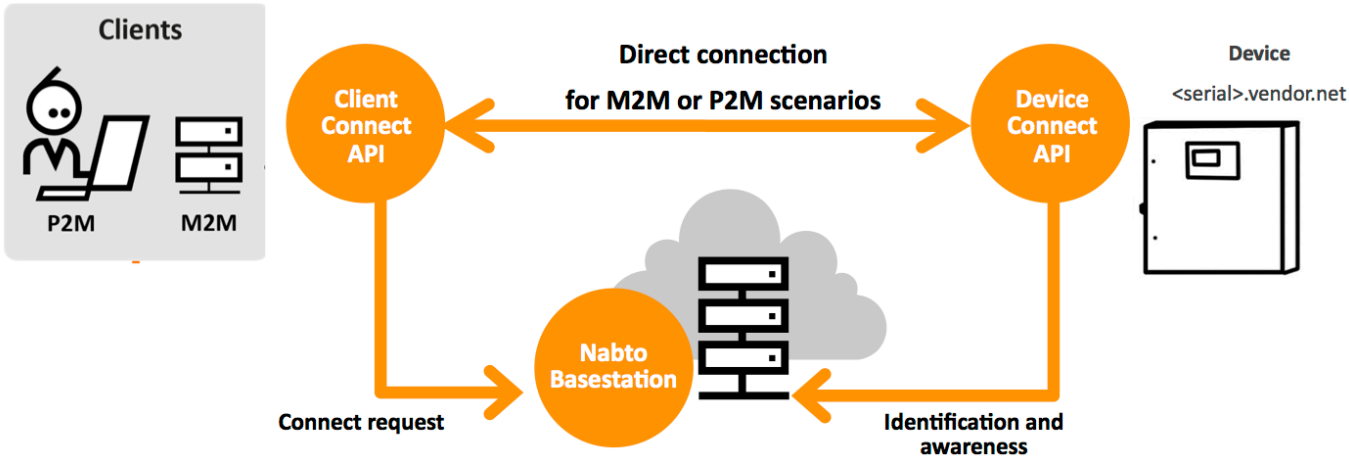
This document describes the client part of a standard uNabto application, which is basically a web application written in HTML and JavaScript using Nabto to communicate directly to a uNabto device through the Nabto Client. The document explains how the HTML application works and it's interface, including some basic examples.

After reading this document an application developer should be able to understand and create a uNabto HTML client application.

2 Bibliography

TEN017	NABTO/001/TEN/017: uNabto SDK - Compiling Source Code
TEN023	NABTO/001/TEN/023: uNabto SDK - Writing a uNabto device application
TEN024	NABTO/001/TEN/024: uNabto SDK - Writing a uNabto HTML client application
TEN025	NABTO/001/TEN/025: uNabto SDK - Writing a Nabto API client application

3 Nabto Platform Basics



The Nabto platform consists of 3 components:

- Nabto **client**: Binaries supplied by Nabto, used by the customer's HTML or native application
- Nabto **device**: The uNabto SDK - an open source framework supplied by Nabto, integrated with the customer's device application
- Nabto **base station**: Services supplied by Nabto (Nabto- or self-hosted) that mediates connections between Nabto clients and devices. Also supplies the user interface to Nabto HTML clients.

The Nabto client initiates a direct, encrypted connection to the Nabto enabled device – the Nabto base station mediates this direct connection: The device's unique name, e.g. <serial>.vendor.net, is mapped to the IP address of the Nabto base station – this is where devices register when online and where clients look for available devices. After connection establishment, the base station is out of the loop – no data is stored on the base station, it only knows about currently available Nabto enabled devices.

The client can also discover the device if located on the same LAN and communicate directly without the base station – useful for bootstrap scenarios or for offline use.

Integrating Nabto on the customer's device is the topic of [TEN023].

The customer's client application may use the Nabto client in different ways: The customer application can be an HTML application that uses the Nabto client to retrieve JSON data in a web application – in this scenario the Nabto client is typically a web browser plugin or mobile app, hosting the customer application. The latter is distributed from the base station to the client and is denoted an HTML device driver bundle. Writing such an application is the topic of this document.

The customer's client application can also be a native (non-HTML) application, linked with a Nabto client API library. The native client application can use the same request/response mechanism to invoke the device as HTML applications do. Additionally, the native client can establish streaming data connections with the device – this is a

popular way of adding seamless, secure remote access capabilities to legacy client and device applications. Native client applications are the topic [TEN025].

4 uNabto HTML applications

The uNabto HTML application is responsible for presenting and handling the user interface in the client browser or app. The application is bundled and distributed in a file called an HTML Device Driver (HTML-DD).

The HTML-DD works the same way as a regular web application (with HTML pages, style sheets layout and JavaScript logic) interfacing JSON with a web service using AJAX (asynchronous HTTP requests). But instead of communicating with a central server using HTTP requests, the Nabto web application communicates internally with the Nabto Client implemented in the browser plugin or app.

The Nabto Client is responsible for connecting and communicating directly to the uNabto device using Nabto's compact UDP peer-to-peer protocol (see figure below). This rather complex connection process is fully transparent for the HTML-DD's point of view – it just sends simple AJAX requests as defined in the common interface and the Nabto Client takes care of the rest.

The common query interface between the HTML-DD and uNabto device is defined in an XML query model file, which is included in the HTML-DD. This query model defines the communication mapped between the HTML/JavaScript and the low level representation on the uNabto device.

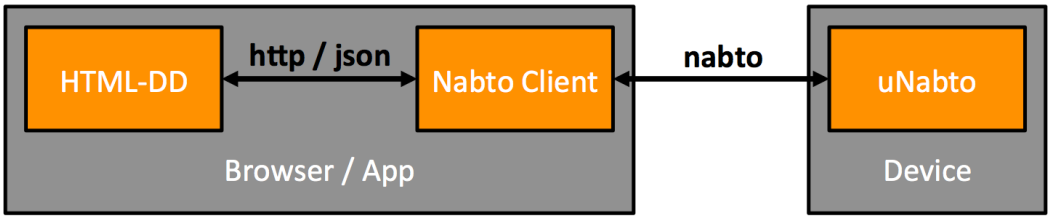


Figure 1: Nabto communication.

The HTML-DD is fetched from a central location based on the connected device's unique ID (if not already installed and up-to-date), see figure below. This has the benefit of customers only having to update the "driver" one place and it automatically pushes it to all clients – without having to submit an update to App Store, etc. It also means that if a customer has already visited a uNabto device, the client does not depend on Internet for the next connection to that device.

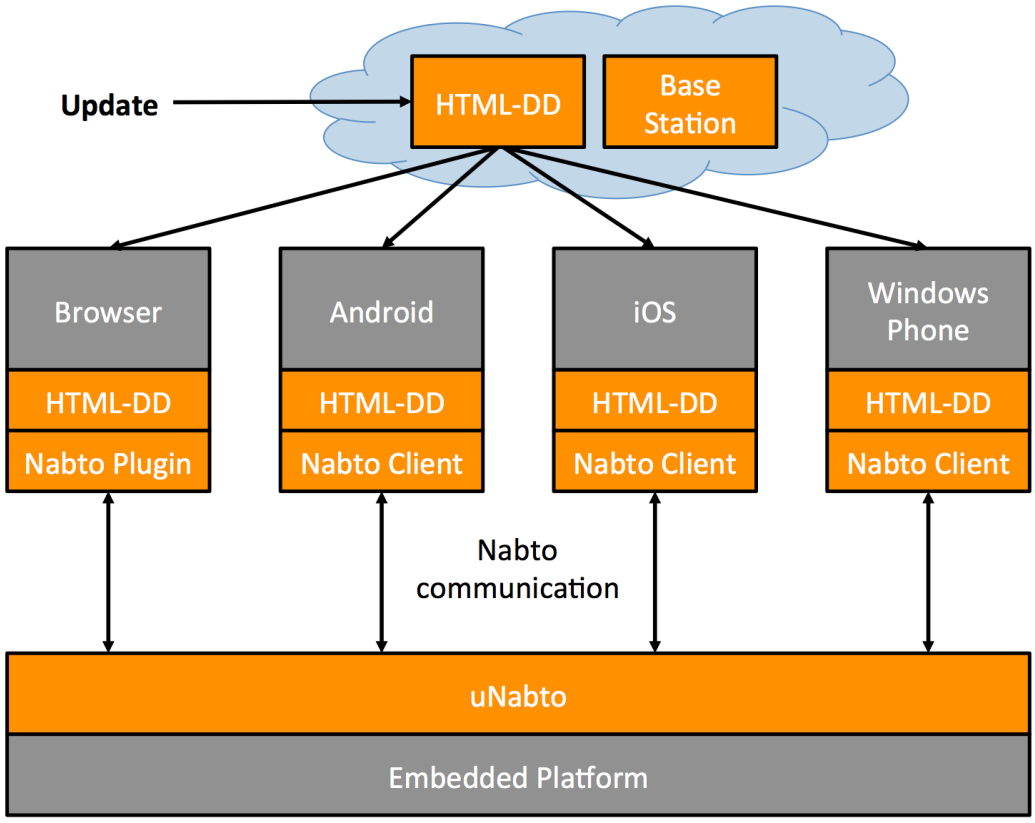


Figure 2: HTML-DD distribution.

The objective when developing an HTML-DD is to put as little logic in the uNabto device as possible. This approach has the benefit of leveraging the heavy computing power of client device nowadays and gives the developers an easier way to update their application from one central location.

The web application approach was chosen due to the ubiquity of web browsers and cross-platform ability. It also gives the application developers a well-known development environment.

5 Nabto Browser Plugin and App

The Nabto Plugin is registered in the browser to serve `nabto://` requests (can be customized in OEM versions to any prefix). Nabto provides a browser plugin for Firefox and Internet Explorer that enables the browser to communicate directly with Nabto enabled devices using the Nabto Client. The HTML-DD is rendered and shown directly in the browser as if the user was viewing a normal webpage.

Because customers often want their own dedicated app for the major phone and tablet platforms, Nabto also provides native applications for iOS, Android and more. These applications work exactly the same as a desktop browser with the Nabto Plugin installed, and basically consists of a full screen web view with a login dialog.

In Firefox and Internet Explorer the Nabto Plugin and asset files are installed into the Nabto base directory (see section “HTML-DD content”).

HTML Device Drivers are often placed on the Nabto Base Station, even though they can be placed anywhere reachable by the browser with `file://` or `http://`. If an installed HTML-DD already exists for the device the client is connecting to, the plugin checks its version against the one on the server. If a newer version exists, it is downloaded and replaces the old. This way the host can update every client's user interface, simple by changing the device driver on the Base Station.

The whole process of checking and downloading HTML Device Drivers on connection is automatically handled by the Nabto Client. When installing an HTML-DD from the Base Station, it is retrieved by the client using the URL format; `http://<host-name>/html_dd`, where `<host-name>` is the ID specified by the user in the URL. The URL also includes query arguments about the client and uNabto device for better selection of an appropriate HTML-DD, based on e.g. browser language. Wildcard DNS records should be used to setup an HTML-DD shared among multiple devices. The full URL format is:

```
http://<host-name>/html_dd?lang=<language>&userAgent=<browser_agent>

&clientVersion=<client_version>&hwVersion=<hardware_version>
```

So for instance the URL `nabto://foo.nabto.net/` could trigger a download of a device driver located at:

```
http://foo.nabto.net/html_dd?lang=en&userAgent=en_EN&clientVersion=16500&hwVersion=2.16300
```

Since the device ID is used to look up the remote device, unique IDs are important. To minimize the risk of name collisions a name like `<mac-address>.<company>.u.nabto.net` is preferred. All uNabto devices named within *u.nabto.net* are configured to connect unencrypted for demonstration purposes.

It is also possible to configure the uNabto device to give a specific URL to the client plugin upon connection, for it's own custom device driver, see *uNabto SDK – How to write a uNabto application*. This makes it easy to give your newly developed HTML-DD to other Nabto users.

`nabto://self/discover` is the Nabto Plugin starting page and is used for local device discovery.

6 HTML Device Driver

The easiest way to communicate with the uNabto devices from the HTML-DD is using XHR / AJAX requests. For that to work the developer needs to add the functionality in the JavaScript logic, which is done even easier with the help of libraries like jQuery.

Using jQuery's JSON[1] or AJAX[2] helper functions:

```
$.getJSON( url [, data ] [, success( data ) ] );
```

```
$.ajax( url [, settings ] );
```

A simple JSON request for data with a `light_id` argument to a uNabto device could look like this:

```
$.getJSON( "read_data.json", {light_id: 1}, function(data) {
      // `data` object now contains the query response, plus original request.
});
```

As jQuery's JSON helper function is just shorthand for it's lower level AJAX helper, the equivalent would be:

```
$.ajax( "read_data.json", {
     dataType: "json",
     data: {light_id: 1},
     success: function(data) {
        // `data` object now contains the query response.
     }
});
```

Most of the Nabto demos use the jQuery Mobile[3] framework to make the HTML work across multiple platforms, ranging from small touch devices to large desktop screens. This framework has some JavaScript touch enhancements and helps with mobile friendly layouts and widgets. Also front-end CSS frameworks like Twitter Bootstrap[4] and Zurb Foundation[5] helps rapid application development.

Because the web applications in Nabto are running from local storage, caching of the requests can be a problem in some browsers. It is therefor recommended to disable AJAX caching in the web application. If using jQuery[6] to

[1] http://api.jquery.com/jQuery.getJSON/

[2] http://api.jquery.com/jQuery.ajax/

[3] http://jquerymobile.com/

[4] http://getbootstrap.com/

[5] http://foundation.zurb.com/

[6] https://api.jquery.com/jQuery.ajax/

abstract away different handling of XHRHttpRequests, then AJAX caching can be disabled with the following command in JavaScript:

```
$.ajaxSetup({cache: false, isLocal: true});
```

6.1 HTML-DD content

The HTML-DD consists of a simple zip file containing the HTML layout material, the JavaScript logic and a description of the data sent between client and device. After the client fetches the device specific HTML-DD from the central location, it is placed in local storage as a completely self-contained web application.

On the client computers, HTML-DD bundles are installed in per-host directories in the `html_dd` sub-directory of the Nabto base directory. The Nabto base directory is `~/.nabto` on Linux and Mac OSX, and `%USERPROFILE%/AppData/LocalLow/Nabto` on Windows.

The following is a tree structure of a typical HTML-DD implementation:

```
html_dd.zip
├── nabto
│   └── unabto_queries.xml        # nabto query scheme
└── static
    ├── index.html                # html page layout
    ├── css
    │   └── style.css                    # style sheet for html
    ├── img
    │   └── image.png                    # image shown on page
    ├── js
    │   └── app.js                # javascript application logic
    └── libs
        └── jquery.min.js         # third-party libraries
```

The `nabto` subdirectory contains the uNabto query model describing the communication queries between client and device.

The `static` subdirectory includes all HTML content and the application logic through JavaScript, just like a normal web application. Typically some third-party JavaScript libraries like the jQuery library are wanted.

6.2 Query model

The query model is the interface between client and device. It is defined in the HTML-DD and describes in detail the requests supported by the device and the responses sent to the client. The Nabto Plugin and Client uses this model directly to parse requests received from the device and encode requests sent to the device.

The query model schema is available from http://www.nabto.com/unabto/query_model.xsd . See *uNabto SDK – How to write a uNabto application* for implementation details on the uNabto device side.

A simple query model example could look like this:

```
<?xml version="1.0"?>
<unabto_queries
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.nabto.com/unabto/query_model.xsd">

<query name="light_read.json" description="Read light status" id="2">
      <request>
        <parameter name="light_id" type="uint16"/>
      </request>
      <response format=”json”>
        <parameter name="light_state" type="uint8"/>
      </response>
</query>
</unabto_queries>
```

This model describes a query named `light_read.json`, with a single input (request) parameter named `light_id` and a single output (response) parameter named **light_state**. The mandatory query nametag is used for identification in the HTML device driver on requests and responses. The mandatory query id attribute is a compact identification of the query, used as opcode when sending requests to the device. The optional description attribute is a user-friendly name for the query. The `format=”json”` in the response, tells the client to convert the response to JavaScript friendly JSON format.

6.3 Query types

The following types are supported in the communication queries:

Types	Description
uint8, uint16, uint32	Common type unsigned integers.
int8, int16, int32	Common type signed integers.
raw	An arbitrary length blob of data.
list	JSON formatted array or dictionary.

The common types **float** and **double** should be sent using raw or list as floating point types vary in representation on different platforms.

A Nabto message must not exceed the maximum UDP packet size limit, which is approximately 1300 bytes when you subtract the Nabto headers (see *uNabto SDK - How to write a uNabto device application* for a complete description of the Nabto packet sizes).

All types received by the HTML-DD are represented as strings in JSON. The client automatically converts requests to the type defined in the HTML-DD's query model, and vice versa with the responses. The type is therefor only really important for the uNabto implementation, while the client side only needs to handle nicely formatted JSON requests / responses.

6.3.1 Raw

The raw type is used for arbitrary length data, e.g. strings, buffers and custom encoded data. When using raw it is left up to the developer to format the request as wanted. Nabto also implements a representation formatter, which can be used to translate the query into hexadecimal format. It is used by supplying the parameter with a `representation="hex"` argument as seen below:

```
<query name="read_data.json" description="Read and write raw data" id="5">
      <request>
        <parameter name="data_read" type="raw"/>
      </request>
      <response format="json">
        <parameter name="data_write" type="raw" representation="hex"/>
      </response>
</query>
```

This changes the JSON output received by the HTML-DD to a hexadecimal representation. E.g. the string "nabto" sent from the uNabto application is received by the HTML-DD as "6E6162746F". The same result goes for the request sent to the uNabto device by appending the hex argument to the request parameter. This method can

become handy when dealing with an embedded application that uses a low level communication protocol like modbus commands.

The following is a simple example of sending a request with a string (raw) to the device and receiving a string response using jQuery AJAX:

```
$.getJSON( "read_data.json", {data_read: "this is a string"}, function(r) {
      console.log(r.response.data_write);
});
```

When receiving JSON formatted raw responses with complex data inside, the Nabto Client automatically escapes evil characters, which would otherwise make it an invalid JSON object. This eases the use of JavaScript libraries like jQuery, because it can parse the response to a JavaScript object without the developer doing anything special for these packages.

6.3.2 Lists

List elements give the application developer the ability to create more dynamic requests and responses. It is the equivalent of the JavaScript object array received in JSON format. A list can contain an arbitrary number of parameters and even other lists.

Here we are expanding on the previous example by making a query that receives a list of ids and returns a list of statuses:

```
<query name="multiple_light_read.json" description="Read light status from given ids" id="3">
      <request>
        <list name="light_ids">
          <parameter name="id" type="uint16"/>
        </list>
      </request>
      <response format="json">
        <list name="light_states>
          <parameter name="state" type="uint8"/>
        </list>
      </response>
    </query>
</unabto_queries>
```

A JavaScript query object for the request could look like this:

```
var data = { light_ids: [{id: 0}, {id: 7}, {id: 42}] };
```

Because jQuery's serialization of input data in `jQuery.ajax` and `jQuery.getJSON` does not work well with arrays, we need to take another approach than the previous mentioned, where the data object is passed directly as an argument to jQuery.

To use lists / arrays in the request, we first need to create the correct request URL by hand, e.g. using this simple approach:

```
var url = "multiple_light_read.json" + "?json=" + JSON.stringify({"request": data});
```

The parameter must be indicated to be encoded in JSON by using "json=". Then the data object must be wrapped in a request and stringified to make it a valid JSON string for the request URL.

The created JSON data string can also be passed in as data to `jQuery.ajax` or `jQuery.getJSON`. Here is a full example seen from JavaScript:

```
var data = { light_ids: [{id: 0}, {id: 7}, {id: 42}] };

$.getJSON("multiple_light_read.json", "?json=" + JSON.stringify({"request": data}),
function(r) {
  // Check if something went wrong.
  if (r.response) {
    // Print the received list to the debugger console.
    console.log(r);
  }
});
```

Returning a JavaScript object like this – note that the client returns the initiating request in the response object for the implementer to verify what parameters was asked for (which is not necessary in this example):

```
{
  "request": {"light_ids": [{"id": "0"}, {"id": "7"}, {"id": "42"}]},
  "response": {"light_states": [{"state": "0"}, {"state": "1"}, {"state": "1"}]}
}
```

For more demonstrations on using lists and raws, see the demo applications in the uNabto SDK.

6.4 Error handling

When writing a custom HTML Device Driver or native Nabto Client application you need to handle possible Nabto Client and application errors.

The Nabto Client reports errors in JSON format, which can be checked in the AJAX success implementation. If something goes wrong with the AJAX request the XHR specific error handler is called. But if something goes wrong with the Nabto AJAX request, the request is still accepted as a success, and the implementer has to check for an error flag in the response. This is typically done by checking if the AJAX response has a response object. If it does not, it must be an error. The same goes for native application implementers.

See "9 Appendix #1: Nabto Error Codes" for format example and complete list of possible error codes.

6.5 How to edit an HTML-DD

To customize a device's HTML-DD in the development phase, the implementor has to prevent his/her custom HTML-DD from being overwritten by the default from the Nabto Base Station. To do this the developer has two choices:

1. Set a flag in the Nabto Browser Plugin that tells the client not to download HTML-DD updates.

Open `nabto_config.ini` located in the the Nabto root directory described in section "HTML-DD content", and change the following:

```
- deviceDriverInstallation=automatic
+ deviceDriverInstallation=never
```

With this solution the implementer can do everything necessary from the client and edit directly in the HTML-DD source, while refreshing the browser to see the changes take effect. Remember to set it back to `automatic` afterwards; otherwise no HTML-DD updates will be received by any visited device.

2. Set a static link in the uNabto device code at compile time, which points to a custom HTML-DD, e.g. a public dropbox link. See *uNabto SDK – How to write a uNabto application* for implementation details.

With this solution every user visiting the device will see your custom HTML-DD and it can easily be tested on phones and tablets.

In production setups the server is set to serve different HTML-DD according to the device ID as described in Nabto Browser Plugin and App.

6.6 Nabto JavaScript Library

Newer HTML Device Drivers are build around a common library suite supplied by Nabto. It is located in the HTML-DD bundle in the `libs` subdirectory. It contains some common third-party JavaScript libraries; jQuery, jQuery Mobile, jQueryFlot, ConversionJS and RequireJS. Most importantly it contains a style sheet that enhances some of the jQuery Mobile styles and the Nabto JavaScript library (nabto.js), which supplies some general helper functions plus Nabto communication and error handling abstractions. The library depends on jQuery and supports AMD loading.

The Nabto JavaScript library is exposed as a global `jNabto` object, which includes all helper functions the implementer can use. It is initialized by calling jNabto.init(options) and takes care of some fundamental functionality, e.g. simple error handling and requests:

- Disable console logging if not present in the browser or debugger.
- Disable AJAX caching.

- Use native notify on supported platform.
- Create a queue for AJAX requests to avoid multiple requests at a time.
- Show message on communication errors.
- Get uNabto and client information.
- Pulls out the correct host path.
- Add connected device to history (recent device list) on discover page.
- Shrink toolbar icons on smaller screens (e.g. discover button).

The Nabto JavaScript library provides the following functions on the `jNabto` object:

- init([options])
  - Initialize the Nabto library with an optional `options` object. See implementation for available option parameters.
- showLogin()
  - Show Nabto login dialog.
- request(query [, callback])
  - Query connected device with relative URL, e.g. `read_button.json`. Optionally with a callback function that includes an error message and response data.
  - `var callback = function(err, data) { console.log(data); };`
- getOptions()
  - Get options set on initialization or defaults at startup.

Below is a small example of using the library from JavaScript:

```
// Query light using Nabto library request helper
function queryLight() {
  jNabto.request("light_read.json?led_id=1”, function(err, data) {
    // If no errors handle response data
    if (!err) {
      console.log(data.light_state);
    }
  });
}

// Initialize on jQuery Mobile page init event and bind event to button
$(document).on("pageinit", "#frontpage", function() {
  jNabto.init({
    debug: true
  });

  $(this).on("tap", "#button", function() {
    queryLight();
  });
});
```

Take a look inside the finished HTML Device Drivers for a more detailed demonstration of using the Nabto JavaScript library.

7 Weather Station example

Look at the Weather Station demo (`nabto://demo.nabto.net/`) for a basic implementation of a HTML-DD. After visiting the device you can find the HTML-DD in the Nabto root directory under `html_dd` with the following tree structure:

```
html_dd.zip
├── nabto
│   ├── unabto_queries.xml
└── static
    ├── css
    │   └── style.css
    ├── index.html
    ├── jquery
    │   ├── jquery-1.10.2.min.js
    │   ├── jquery.mobile-1.3.2.min.css
    │   └── jquery.mobile-1.3.2.min.js
    └── js
        ├── defaults.js
        └── helper.js
```

The `unabto_queries.xml` query model defines three queries for communicating with the uNabto device:

- `house_temperature.json`

  Takes a `sensor_id` in uint32 format and respond with a `temperature` in uint32 format.

- `wind_speed.json`

  Takes no arguments and responds with a `speed_m_s` in uint32 format.

- `weather_station.json`

  Takes no arguments and responds with `temperature, windspeed` and `humidity` data.

`index.html` of the web application, located in the `static` subdirectory, includes the jQuery and jQuery Mobile frameworks, a custom stylesheet, defaults.js, helper.js and the initial layout in HTML using the jQuery Mobile syntax.

`js/defaults.js` sets the needed options for jQuery Mobile before the framework is loaded, e.g. disable caching and default transitions. And it makes sure the console object is defined to avoid errors in older browsers.

`js/helper.js` contains the application logic. It uses jQuery to bind to each button and send the corresponding query to the device using getJSON:

```
$.getJSON(request, function(r) {
  $(".errors").hide();
  if (r.response) {
    action(r.response);
  }
}).error(function (error){
  $(".errors").show();
});
```

8 Nabduino example

For a more complex demonstration of what an HTML-DD can do, look at the Nabduino demo (`nabto://demo1.nabduino.net/`). This implementation uses a multiple page layout with multiple functionality and the Nabto JavaScript library in `libs/`.

Look at the implementation for further details.

9 Appendix #1: Nabto Error Codes

Nabto error codes are divided in two scopes; 1.000.000 for fundamental Nabto errors and 2.000.000 for general application errors. These error codes and short description is delivered in JSON format as shown below:

```
{
    "error": {
        "event": "2000039",
        "header": "File Not Found (2000039)",
        "detail": "nabto://foo.demo.nab.to/unexisting.json?",
        "body": "The requested file could not be found. URL of requested file was
nabto://foo.demo.nab.to/unexisting.json?"
    }
}
```

The special error code 2000065 (UNABTO_APPLICATION_EXCEPTION) allows the device application to return a specific error code from the application_event handler - meaning that all communication between the devices was ok, but some exceptional condition occurred at the application level in the device.

The possible error codes to use in the device and their mapping to what the client sees can be found in the uNabto SDK (`unabto/src/unabto/unabto_app.h`) . See [TEN023] section "The client query handler" for details on the return errors in the device application.

The error is supplied to the application in the "detail" field:

```
{
    "error": {
        "event": "2000065",
        "header": "Error in device application (2000065)",
        "detail": "NP_E_INV_QUERY_ID",
        "body": "Communication with the device succeeded, but the application on the device
returned error code NP_E_INV_QUERY_ID"
}
```

If available, the Nabto Client Log can be inspected for more details. Or remote syslog can be enabled from the basestation for both devices and clients, contact Nabto support for more information in this regard.

9.1 Error Code List

The following is a complete list for Nabto error codes:

1000000: **UNSPECIFIED_ERROR**

1000001: **PROGRAM_VERSION_CONFLICT**

1000002: **PROTOCOL_VERSION_CONFLICT**

1000003: **UDP_SOCKET_CREATION_ERROR**

1000004: **RESOLVER_ERROR**

1000005: **STUN_ERROR**

1000006: **UDT_SOCKET_CREATION_ERROR**

1000007: **UDT_CONNECTION_ERROR**

1000008: **FALLBACK_CONNECTION_ERROR**

1000009: **CONNECTION_ERROR**

1000010: **UNKNOWN_SERVER**

1000011: **ACCESS_DENIED**

1000012: **CANNOT_VERIFY_CLIENT_CERTIFICATE**

1000013: **BAD_ID_IN_SERVER_CERTIFICATE**

1000014: **CANNOT_VERIFY_SERVER_CERTIFICATE**

1000015: **MICROSERVER_NOT_KNOWN**

1000016: **CONNECTION_PROBLEM**

1000017: **ATTACH_LOST**

1000018: **RESSOURCE_PROBLEM**

1000019: **SYSTEM_ERROR**

1000020: **ENCRYPTION_MISMATCH**

1000021: **MICROSERVER_BUSY**

1000022: **MICROSERVER_ADDR_MISMATCH**

1000023: **NO_RSP_FROM_CONTROLLER**

1000024: **MICROSERVER_REATTACHING**

9.2 Certificate Handling

1000100: CERT_ID_NOT_FOUND

1000101: CERT_FILE_NOT_FOUND

1000103: CERT_INVALID_PSW

You could not be logged on to the network. Please verify that you specified an email address and password that matches your profile on the central server.

1000102: CERT_FILE_INVALID

1000104: CERT_KEY_FILE_MISSING

You could not be logged on to the network due to problems related to accessing your identity information on this machine.

9.3 General Application Errors:

2000000: UNSPECIFIED_APL_ERROR

2000001: HTTP_SERVER_UNAVAILABLE

Connected to device but no service available on device. Please make sure the correct software is running on the target device and it is possible to reach the forward destination (for instance, check that the Apache HTTP server running on the remote host).

2000002: MISSING_PARAMETERS

A parameter was not specified in input.

2000003: PASSWORD_TOO_SHORT

2000004: DATA_DIR_ACCESS_ERROR

Could not Access Data Directory.

2000005: CERT_CREATION_ERROR

Could not Create Certificate.

2000006: CERT_SIGNING_ERROR

Error Signing Certificate.

2000007: CERT_SAVING_ERROR

Error Saving Certificate.

2000008: HTML_TEMPLATE_RENDERING_ERROR

2000009: PORTAL_LOGIN_FAILURE

2000010: PROXY_COULD_NOT_START

The client could not start its proxy server.

2000011: NETWORK_PROBED

2000012: POST_DATA_RETRIEVAL_FAILED

2000013: POST_DATA_SUBMISSION_FAILED

2000014: NOT_LOGGED_IN

2000015: TIME_OUT

The operation did not complete within the configured time limit.

2000016: INVALID_PORT_NUMBER

Invalid tunnel TCP port number specified.

2000017: TUNNEL_COULD_NOT_START

The tunnel could not be started, please see log file for details.

2000018: PORT_ALREADY_IN_USE

The specified port is already in use, please choose another one.

2000019: TUNNEL_DESTRUCTION_ERROR

The tunnel could not be stopped, please see log file for details.

2000020: DEVICE_REGISTRATION_FAILED

An error occurred when registering device.

2000021: COULD_NOT_UNLOCK_DEVICE_KEY

Invalid password specified, key file corrupt.

2000022: COULD_NOT_STOP_CLIENT

2000023: QUERY_MODEL_PARSE_ERROR

Could not parse the query model file, unabto_queries.xml.

2000024: INITIALIZATION_ERROR

2000025: INTERNAL_ERROR

An internal error occurred in the client software.

2000026: QUERY_MODEL_INVALID_ID

2000027: QUERY_MODEL_NO_SUCH_REQUEST

2000028: QUERY_MODEL_NO_SUCH_PARAMETER

2000029: QUERY_MODEL_PARAMETER_PARSE_ERROR

2000030: QUERY_MODEL_MISSING_PARAMETER

2000031: GUIREP_DOWNLOAD_FAIL

Could not retrieve HTML Device Driver.

2000032: QUERY_SEND_FAILURE

2000033: QUERY_RESPONSE_RECV_FAILURE

2000034: QUERY_RESPONSE_DECODE_FAILURE

2000035: NO_ACTIVE_REQUEST

2000036: UNEXPECTED_RESPONSE_SIZE

2000037: GUIREP_INSTALL_FAILED

2000038: GUIREP_BAD_STRUCTURE

2000039: FILE_NOT_FOUND

The requested file could not be found.

2000040: INSTALLATION_FILE_MISSING

A file is missing from the installation directory, please re-install the software. See log file for details and contact support if the problem persists.

2000041: PORTAL_AND_KEY_PASSWORD_MISMATCH

2000042: INVALID_URL

Could not parse the URL.

2000043: OTHER_TASK_ACTIVE

The request could not be completed as another request is already active.

2000044: UNSUPPORTED_OBJECT_TYPE

2000045: TCP_SOCKET_PROBLEM

2000046: ROOT_CERT_MISSING

2000047: CONNECTION_RESET_BY_PEER

The remote end has closed the connection.

2000048: NO_NETWORK

2000049: NO_INTERNET_ACCESS

2000050: LOCAL_MICRO_CONNECT_FAILED

2000051: INVALID_EMAIL_ADDRESS

2000052: EMAIL_ADDRESS_IN_USE

2000053: DEVICE_ERR_UNKNOWN_QUERY_ID

The device did not recognize the specified query.

2000054: BUFFER_TOO_SMALL

2000055: INVALID_BUFFER

2000056: INVALID_ENDPOINT

2000057: NO_DATA_AVAILABLE

2000058: DATA_TRANSMISSION_PROBLEM

2000059: SESSION_KEY_MISSING

The session key is not present, and the security configuration for the html device driver requires a valid session key.

2000060: SESSION_KEY_INVALID

The session key given in the request is not valid.

2000061: MANIFEST_PARSE_ERROR

2000062: GENERIC_LOGIN_FAIL

2000063: QUERY_JSON_PARSE_ERROR

JSON format in the request/response is invalid.

2000064: EMPTY_PARAMETER

2000065: UNABTO_APPLICATION_EXCEPTION

Communication with the device succeeded, but the application on the device returned error.

2000066: QUERY_MODEL_MISSING

The file unabto_queries.xml is missing or unreadable.