Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Deliverable 1 - Part 2

IoT Remote Lab, SS 2020

M. Sc. Ege Korkan

- For all the questions below, make sure to work in a git directory. We evaluate how well you use git, meaning working in branches, not committing everything in one go, meaningful commit messages etc.
- Each file needs to be named the way they are specified in the questions. If the file name is wrong and we cannot find it, you will get no points. **For all the questions in the sheet, create a directory or folder called D1_2. All the following locations take this as root directory.**
- You are encouraged but not obliged to have a good README.md file that explains the directory structure.

? Exercise 1.1: Partial TDs

In this exercise, you will be writing parts of a TD that are not valid TDs. Do not try to validate them in the TD Playground. The files you should create should always contain the `p_` prefix to denote that they are partial TDs and have the `.json` ending. The TDs should contain only the parts that are asked, we will validate with `"additionalProperties":false`. Bonus questions bring you points only if you get less than 100% in the first deliverable, i.e. they do not get transferred to upcoming deliverables. All the questions are independent of each other, you might answer some of them easier after thinking about the next ones. If you need to write a URI, you can simply use `example.com` as the base URI and choose your favorite protocol with a URI scheme.

1. We will start with vocabulary terms that are **not** Interaction Affordances.
    1. Write a TD file called `p_version.json` that denotes that this TD is the version `10.21.2`.
    2. Bonus: According to semantic versioning (https://semver.org/), match the following changes in a Thing to MAJOR, MINOR and PATCH numbers. For your answer, create a file called `b1.json` which is an object that has keys `"1"`,`"2"` and `"3"` which can have values `"MAJOR"`, `"MINOR"` and `"PATCH"`.
        (a) Thing and its TD gets a new Interaction Affordance.
        (b) A Property Affordance that is supposed to return a `number` was returning a `string`. This gets fixed in a new version.
        (c) An Action Affordance that was expecting an `object` as an input changes to expect a `number`.
    3. Write a TD file called `p_noSec.json` that declares that all Interaction Affordances can and do use only no security, i.e. anyone in the correct network can invoke actions, read properties etc.
    4. Write a TD file called `p_basicAuth.json` that declares that all Interaction Affordances can and do use only Basic Authentication where the credentials are sent in the header.
    5. Bonus: Provide the cURL request for the URI `https://example.com` with an HTTP GET request where the credentials are `"hello"` for the username and `"world"` for the password. Name it `basicAuth.sh`.
    6. Write a TD file called `p_links.json` that describes two links to this TD. One refers to a resource that is the subject or topic of the link's context, i.e. provides more information and the other one signifies that the IRI in the value of the `href` attribute identifies a resource equivalent to the TD, i.e. signifies the TD itself.

7. TDs must contain a `title` key that gives the TD a name and it can contain a `description` that gives more detailed and descriptive information. Write a multi language TD that has the default language of English but also has German support. The title in English should be `"Robot"`, in German `"Roboter"`, the description in English should be `"A robot on a moving platform"`, in German `"Ein Roboter auf einer mobile Plattform"`. Name it `p_multiLangEn.json`
8. Bonus: Write the same TD as above but the default language is German instead of English. Name it `p_multiLangDe.json`

2. Now Interaction Affordances! However, we will **not** go into the forms part of the affordances, i.e. do not write the forms.

   1. Write a TD file called `p_properties.json` that has two properties called `targetSpeed` and `currentSpeed`. `targetSpeed` can be written to by the Consumers whereas `currentSpeed` can be only read. They both represent `number` values between 0 and 120 and have units of `"km/h"`. Additionally, `currentSpeed` can be also observed.
   2. Write a TD file called `p_actions.json` that has 3 Action Affordances.
      - `goHome` does not require any input and does not return an output payload. It moves a robot to a pre-programmed location no matter the input.
      - `goTo` takes an input object with `x`, `y` and `z` keys where all the keys of number type must be present. `x` is between 0 and 180, `y` between 0 and 90, `z` between 0 and 20. It does not provide an output and goes to the same location from any initial state.
      - `rotate` rotates the base of the robot relative to its initial position based on the input and returns the final location. The input is a number between 0 and 360 and the output is an object with `x`, `y` and `z` keys where all the keys must be present.
   3. Bonus: For the `rotate` action of the previous question, what if the Thing provides in XML format? Write the TD in this case, name it `p_actionsXML.json` and provide an example payload, named `xmlPayload.xml`.
   4. If the `goHome` action does not provide an output how does the Consumer know that the action has finished? Formulate a human readable answer in a paragraph of maximum 4 sentences in Markdown format. Name it `noOutput.md`.
   5. Write a TD file called `p_events.json` that has 2 Event Affordances:
      - `emergencyButton` does not deliver any payload.
      - `position` takes a number as the frequency the robot will push the position data. Then it delivers the event data as an object with `x`, `y` and `z` keys where all the keys must be present.

? Exercise 1.2: TDs

In this part, you will need to write entire TDs that need to be valid according to the TD Playground[1] which validates according to the official TD Validation Schema[2]. If your TD is validated, it does not mean that it is the answer of a question. Make sure that `(With Defaults) JSON Schema` lights green in the TD Playground.

The first TD is for the robot that you already got familiar with and the second one will be a slight variation of it. The third TD will be for a system controller that controls two such robots. Since the lectures on the protocols are not done yet, you will not graded on the correct use of the protocol methods, i.e. if you say that an action is invoked with an HTTP GET, it is fine.

[1] http://plugfest.thingweb.io/playground/
[2] https://w3c.github.io/wot-thing-description/#json-schema-for-validation

1. Write a TD and name it `robot1.json`. The robot has an end effector that can be moved to objects and grab them. It should:
   - have `title` and `description` of your choice.
   - have no security mechanism
   - use HTTP for all the interaction affordances with a common base URI for all interactions. The base URI should be `http://example.org/robot1`
   - have a property called `location` to read and observe the location of the robot's end effector. It delivers an `object` in JSON media type with `x`, `y` and `z` keys where all the keys of number type are always present. `x` is between 0 and 180, `y` between 0 and 90, `z` between 0 and 20. Observation requires a subprotocol called `longpoll` on a different URI than reading.
   - have a property called `state` that can be read and written to that can have string value of `enabled` or `disabled`
   - have a property called `updateFrequency` that can be read and written to that contains information on how frequently the robot updates the internal end effector location data from its sensors. It is a `number` in JSON media type who has `Hertz` unit and can have the values of `1`, `10` or `100`.
   - allow all three properties to be read in a single request
   - have the actions in question 1.2. Their input and output are in JSON media type.
   - has an event called `emergencyButton` that does not deliver any payload.
   - has an event called `limitHit` that can notify the Consumers when the end effector reaches one of the limits while moving. The payload is in JSON media type and is an object with two keys called `axis` and `direction`. `axis` can be of string values `x`, `y` or `z` and `direction` can be strings of value `positive` or `negative` for `x` and `y` axes and can be only `positive` for the `z` axis.
2. Write a TD and name it `robot1C.json` that is very similar to the previous question. You can simply copy paste it and change the necessary fields. The differences are:
   - It uses Bearer Token as a security mechanism where the security authentication information is passed in the header.
   - It has two base URIs where first is bound to HTTP, like in the previous question but the other one uses CoAP as the protocol and CBOR for all the media types. The base URI for the CoAP is `coaps://192.168.0.101:5683/robot1`
3. In this question, think of a controller that uses two robots of the question 2.1 as part of a production chain. Its TD is very simple but we want to have a sneak peek into how its internals would need to work. The second robot has the base URI of `http://example.org/robot2`.
   1. Write a TD and name it `controller.json`. It should:
      - have `title` and `description` of your choice.
      - have Basic Authentication as a security mechanism where the security authentication information is passed in the header.
      - use HTTP for all the interaction affordances with a common base URI for all interactions. The base URI should be `http://example.org/controller`
      - have actions called `startSystem` and `stopSystem` that internally enables and disables the robots using their `state` properties. They take no input or provide output.
      - have an action called `action1` that internally moves robot1 to its home position and moves the robot2 to position `x:1,y:10,z:0`. It takes no input or provide output.

4. Write three JSON files that are of array type called `startSystem.json`, `stopSystem.json` and `action1.json`. They should contain the forms (JSON Objects) of robot1 and robot2 (you should copy paste) that are of relevance for the action in the filename. The order does not matter. The payloads that are sent should not appear in these arrays!