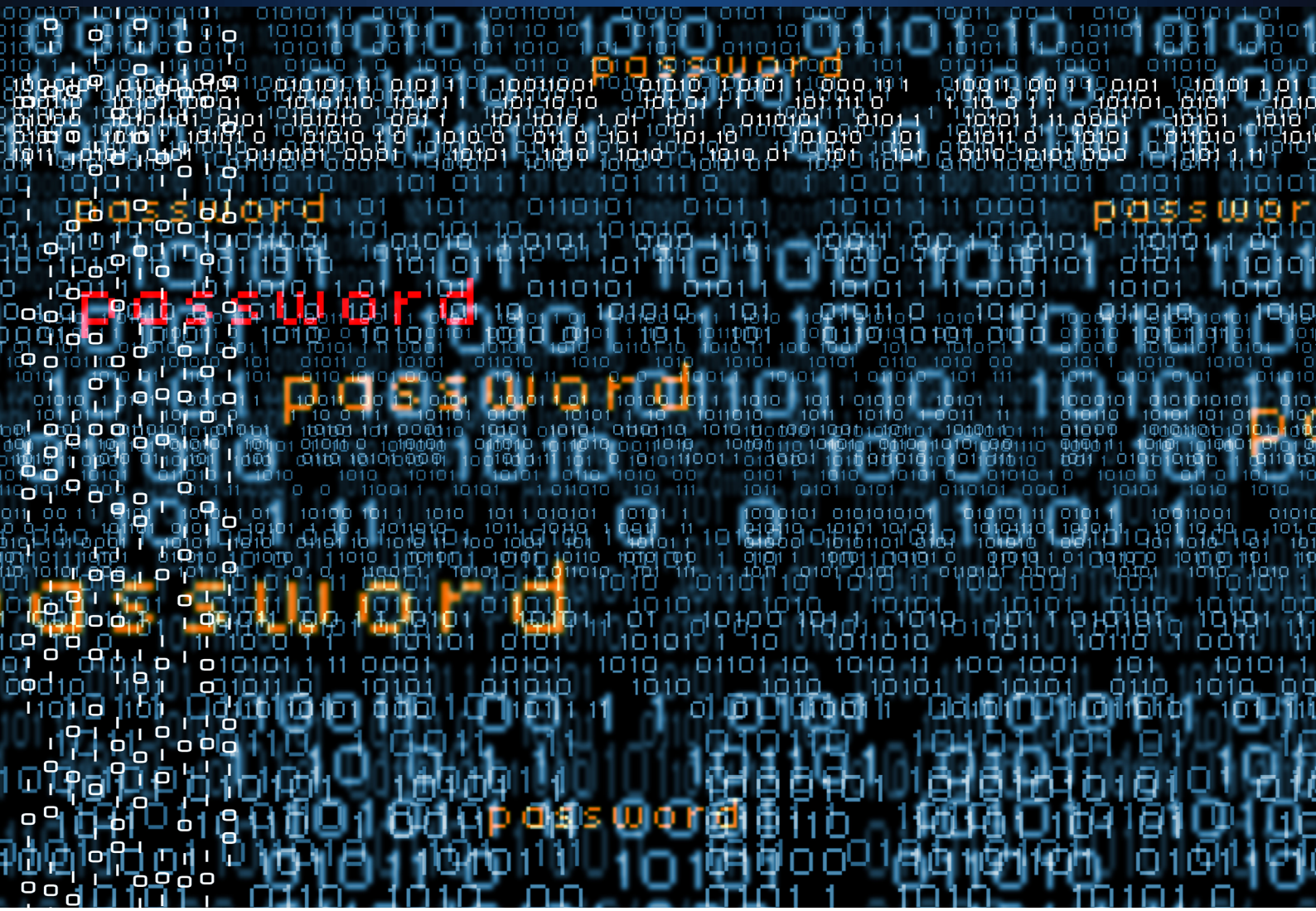
iBRoad

Individual Building
Renovation Roadmaps



Database structure and programming core of the roadmap and logbook

Blue Planet Academy & Consulting

November 2018

www.ibroad-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 754045

Authors (BPAC)

Michaël Van Damme

External reviewers

BPAC would like to acknowledge the contribution of iBRoad partners in reviewing this report, in particular TU Wien and ifeu.

Published in November 2018 by iBRoad.

TABLE OF CONTENTS

I. INTRODUCTION

This deliverable describes the data structure and current software implementation of the iBRoad Roadmap Assistant and Logbook components.

II. DATA STRUCTURE

The iBRoad tool consist of two components: the Roadmap Assistant and the Logbook, both of which are implemented as web applications.

The data structure for the Roadmap Assistant can be considered to be "standard": the data fields it needs to store are fixed (i.e. determined in advance), and the data fits the model of a relational database (where data is organised in different tables linked to each other) very well. Hence, the requirements posed by the data structure are very similar to what is required by the large majority of web applications, and a typical relational database is perfectly suitable.

The situation for the Logbook is different, however, as explained below.

i. Logbook data structure

iBRoad partner ADENE has suggested a detailed and well-thought-out concept for the logbook structure, summarized in figure 1. It consists of both a fixed part (levels 0, 1 and 2), which lists all topics every national (or regional) logbook should contain, and a flexible part (levels 3 and higher), which is country or region specific. This document explains how this concept can be implemented in an IT system.

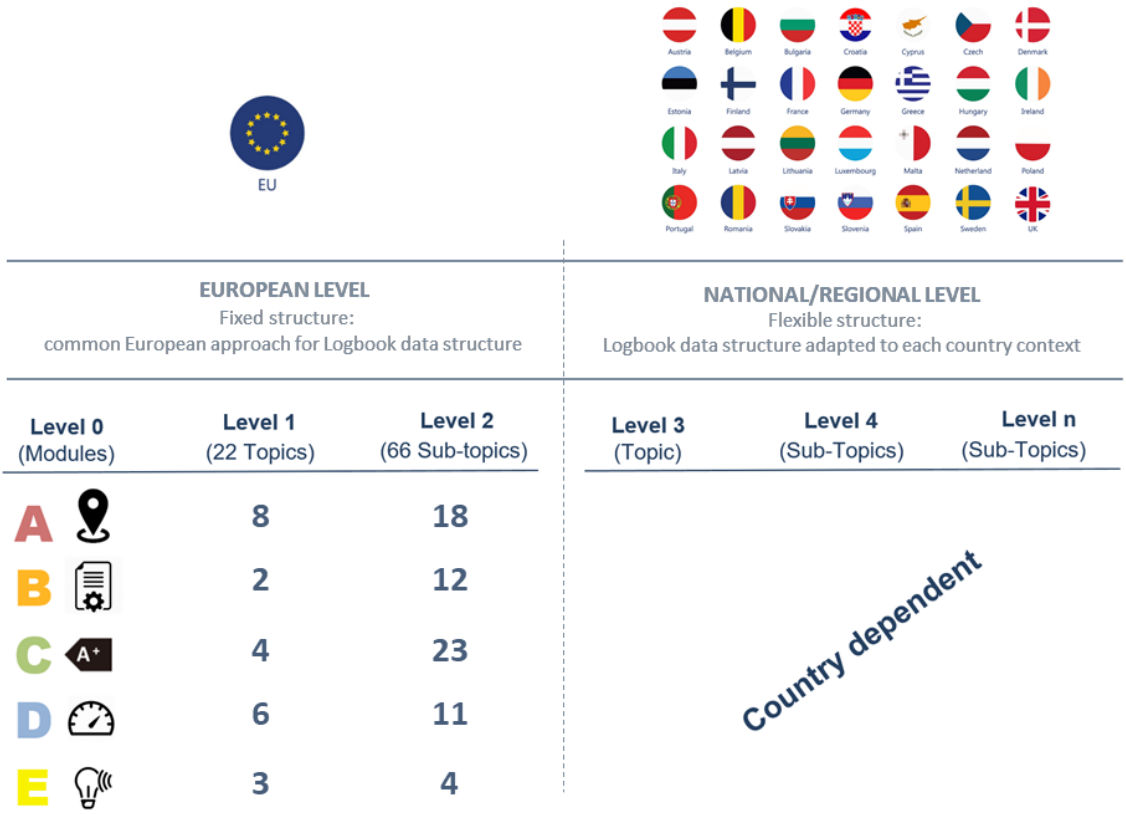


Figure 1: *Logbook structure concept (source: "iBRoad - The logbook data quest" by ADENE)*

a. Flexible data structure

In order to understand the challenges involved in implementing the logbook structure in a database, let's consider an example. Table 1 shows how the above structure concept foresees to store information about a building's age in a logbook for Portugal. Since everything up to level 2 is fixed (and hence always present), table 1 shows that every country or region is required to store information about the sub-topic "Building", belonging to topic "Building general features", which is part of module "A. General and administrative information". For Portugal, the choice was made to include (among other items) information about the building's age (level 3), split into two fields (level 4): year of construction and year of renovation.

Level 0	Level 1	Level 2	Level 3	Level 4
A. General and Administrative information	...			
	4. Building general features	1. Building	...	
			3. Building age	1. Construction year
				2. Year of renovation
			...	

***Table 1**: Example – a building's age in the logbook for the case of Portugal*

In a relational database (by far the most common type), data is organised in (usually multiple) tables consisting of columns and rows. Roughly speaking, the database structure consists of a description of all tables and their columns (column names and data types). In virtually all cases the database structure is fixed (i.e. the tables and columns are defined in advance), the data itself is then placed in the rows of the tables by the application.

If we would design a database specifically for Portugal, the information in table 1 would lead us to create e.g. a table called "Buildings"[1], containing columns "year of construction" and "year of renovation"[2] (among others). For each building stored in the logbook, we would add one row to the "Buildings"-table, with the construction and renovation years stored in the relevant columns.

The iBRoad logbook should not only work for Portugal, however, but for any country or region wishing to implement it. Due to the flexibility built into the logbook concept we cannot define the database columns as described above, since each country/region should be able to choose its own columns (because all

1 In order to avoid any confusion, please note that this name is not related to the level 2 parameter "Buildings" in the example, it simply is the table in which we store all building-related information.

2 Note that only the nodes at the highest level for each row in a table such as table 1 would result in a database column.

parameters at level 3 and higher are country specific). This is the first major challenge: how to combine the inherently fixed and predefined structure of a relational database with the need for high country/region specificity in determining what data to store?

We decided to introduce flexibility using JSON-fields[3]. In the unmodified original proposal, this would work as follows: all level 2 sub-topics become database columns. Inside these columns, we put all parameters belonging to the relevant sub-topic in a country specific JSON data structure. If we take the example of table 1, we would create a column called "building" (named after its level 2 sub-topic). For each row (i.e. for each building in the logbook), the contents of that column would be a JSON-structure containing the parameters to be stored under this sub-topic for that country / region. For Portugal, the relevant part of the JSON (i.e. only the part related to building age) would look something like this for a specific building:

```
{
   "Building age":{
      "Construction year":1967,
      "Renovation year":2003
   }
}
```

Another country might choose different parameters for the same sub-topic, resulting in the following JSON content, for instance:

```
{
   "Age of the building":{
      "Date of completion":"1967/5/23",
      "Renovations":{
        "Technical Building Systems": 2003,
        "Structure": 1998
      }
   }
}
```

The structure of the JSON for each sub-topic is described in a configuration file (one file for each country or region). This description does not influence the data structure but is needed for the user interface, since it determines which information a user must or can enter.

In the last few years most relational database systems (including PostgreSQL, the database system selected for iBRoad) have added native support for JSON columns, which implies that database queries can also operate on keys within JSON fields.

3 JSON is a commonly used human-readable data interchange format.

b. Hybrid Data Structure

The above described proposal offers a lot of flexibility, but this flexibility comes at a price. Since most data items are contained in JSON columns, the data is largely unstructured, and the logbook application does not really "understand" what data is being stored, or how to use it. This lack of structure makes it difficult to implement logbook features which depend on the data entered by the user.

We will consider colour coding as an example to illustrate this. Suppose we want to color-coded certain elements in the user interface based on (among others) the year of construction of the building. Of course, this means the application must be able to extract it from the entered data. For the first JSON listing shown above, we could get this data as follows:

```
"Building" (JSON-column)  -> "Building age"
                          -> "Construction year"
```

For the second listing, it would be:

```
"Building" (JSON-column)  -> "Age of building"
                          -> "Date of completion"
                          -> Operation: Extract year from date
```

The problem is that it is not clear in advance how to extract this value: it depends on the choice of parameters made by the country, it might require one or more operations or transformations (in this example, extracting the year from a date is a simple illustration of this), it might not even be present if the country chose not to include it.

A similar example is a (simplified) energy demand calculation, which would require U-values for all walls (and of course many other parameters). The level 2 parameter "Walls" is present in the original logbook structure concept, so every country is required to store information about walls, but since all higher levels are country specific, there is no general way of knowing what countries will choose to store, or how to extract the needed U-values from that.

This is the second challenge: how do we combine logbook features that depend on data with the need for a flexible data structure? Unfortunately, adding more structure to the data seems to be the only workable solution. This is why we chose to make the "boundary" between what is fixed (normally everything up to level 2) and what is flexible (normally levels 3 and up) subject dependent. This allows imposing more structure when necessary (and only then) by selectively increasing the number of "fixed" levels. For walls, for example, we could require everything up to level 4 to be fixed, and place the parameters we strictly need to know for calculations at that level. Any additional information about walls (which can be country specific) can be placed at level 5 (or higher). Of course, this only makes sense for parameters for which we can objectively assume that they will be present and meaningful regardless of the country (such as U-values of walls, years of construction of a building, etc.).

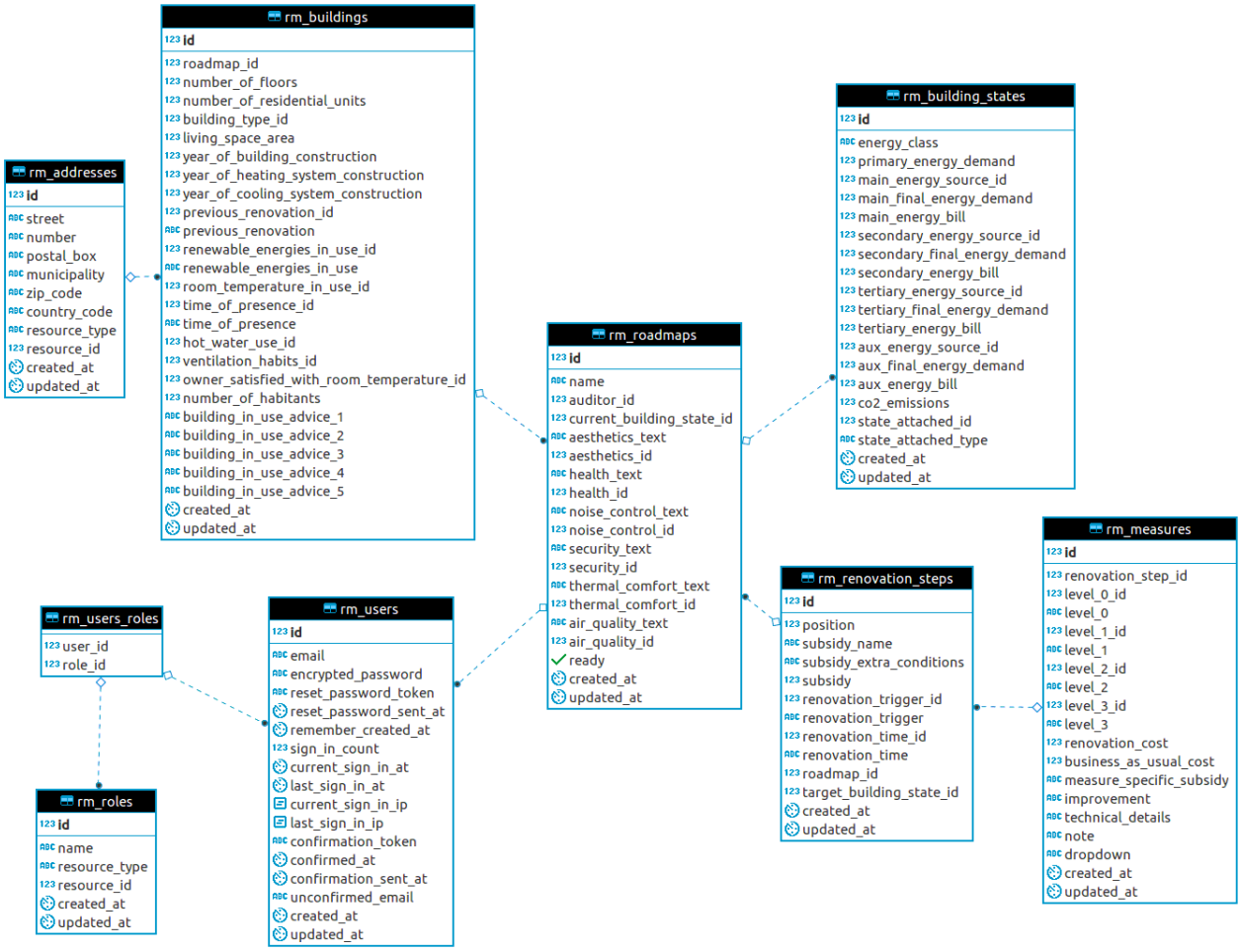
This is what we call the "hybrid" data structure: it isn't as flexible as the above described flexible data structure, but there are still a lot of possibilities for adding country/region specific information, so it isn't fixed either.

Note that due to the fact that the number of fixed levels will probably often be increased for entities of which we may have many (such as wall types (the user can define as many wall types as necessary), roof types, floor types, etc.), the resulting database structure of the hybrid structure is fundamentally different (and more complicated) than for the fully flexible data structure.

ii. Data structure diagrams

a. Roadmap Assistant

The database structure for the Roadmap Assistant is shown below in the form of an Entity Relationship Diagram.



The main table is "rm_roadmaps"[4], which groups the information for each roadmap. A roadmap is linked to a user (table "rm_users"), which can itself have different roles (table "rm_roles"). The many-to-many relation between users and roles is realised by means of the "rm_users_roles" join table.

A roadmap applies to a single building ("rm_buildings" table) and is linked to several building states ("rm_building_states" table) which contain information on energy use before (original state) and after (new state) renovation, for instance.

A roadmap consists of renovation steps ("rm_renovation_steps" table), each encompassing a number of more concrete renovation measures ("rm_renovation_measures" table).

4 All tables related to the Roadmap Assistant component have the prefix "rm" (shorthand for roadmap) in their names.

b. Logbook

The database structure for the Logbook is shown below conceptually (i.e. without full details for each table) in the form of an Entity Relationship Diagram.

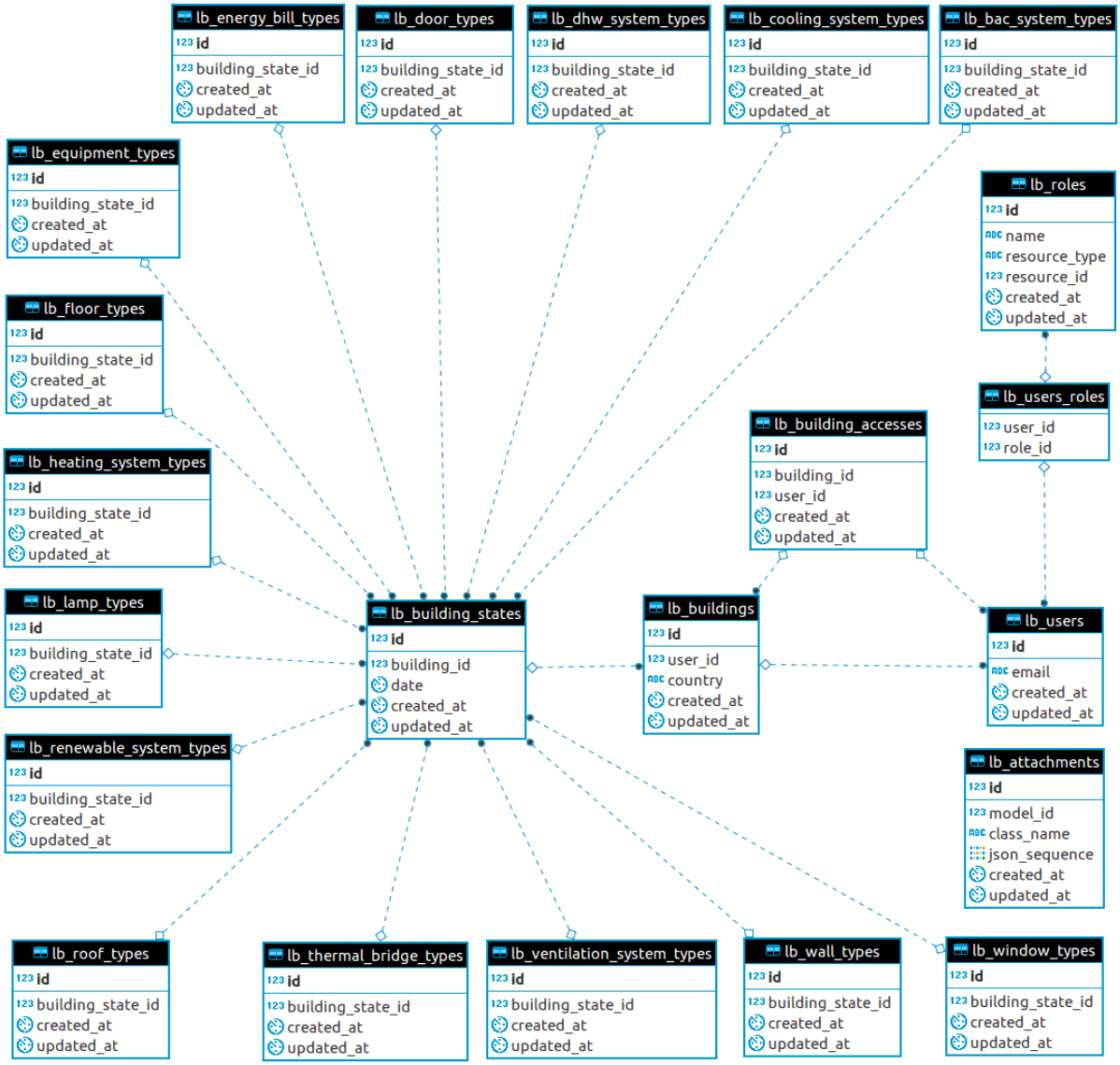
The users and roles tables ("lb_users", "lb_users_roles" and "lb_roles")[5] are identical to the corresponding tables in the Roadmap Assistant.

A user can own buildings ("lb_buildings" table) and can in some cases (e.g. auditors) also have access to buildings he does not own by means of the "lb_building_accesses" join table.

A building can have several building states, stored in the central and most important table in this data structure, the "lb_building_states" table. In the context of the logbook, a building state is the central repository of information for the building at a certain time, and is linked to all the tables with technical data shown around it, with information about wall types ("lb_wall_types"), window types ("lb_window_types"), etc. The idea is that all information about a building is entered and linked to the "current" building state. When the building is then renovated (or otherwise modified), a new building state is created, into which all data for the renovated building can be stored. This way, all information regarding previous "states" of the same building is preserved and remains available.

The "lb_attachments" table is used for storing information about uploaded documents, plans, photo's, etc. It isn't shown as explicitly linked to any other table, since each row (record) can be linked to a record in a different table (polymorphic association).

5 All tables related to the Logbook component have the prefix "lb" in their names.



iii. Source Code

a. Ruby on Rails

Both Roadmap Assistant and Logbook are web applications written in Ruby on Rails. Ruby on Rails (often just called Rails) is an MVC (model-view-controller) server-side web-framework written in Ruby, an object-oriented dynamic programming language.

By default, every model class in Rails maps to a database table, with objects of the class corresponding to individual records in the table (this is an application of the object-relational mapping (ORM) technique). Controllers are classes that coordinate: they receive requests made to the application (e.g. as a response of clicks in the browser), load the models and interact with them, and determine which so-called view will be rendered and returned as a response.

Rails is a so-called “opinionated” framework, which means that it provides default choices for almost everything that is configurable. Hence, Rails applications have a standardized structure known to every

developer familiar with Rails, which makes it much easier to get accustomed with an unknown application's source code.

Both applications use PostgreSQL (version 9.4 or higher) as the database, a very established open-source database with excellent support for JSON fields.

b. Roadmap Assistant

The source code for the Roadmap Assistant is annexed (digitally) to this deliverable. It is a fully functional and almost complete, although it is probable that some details will be modified, or features added during the rest of the project.

All database tables shown above have a corresponding model class. Various controllers and views, complemented by front-end code written in Ruby (technically in Opal) and transpiled to Javacript create a user interface for auditors to edit all aspects of an iBRoad Roadmaps (renovation steps, renovation measures, building states, etc.).

The Roadmap Assistance is in all aspects a standard Rails application that can be understood by any experienced Rails developer.

c. Logbook

Contrary to the Roadmap Assistant, the Logbook application is not finished yet. The current version of the source code is annexed digitally to this deliverable (the code is runnable, but with an unfinished interface and very limited functionality).

Apart from being a much larger application than the Roadmap Assistant, several aspects make the Logbook more complicated:

- An important design goal for the Logbook was to have as little code as possible that is country or region dependent and handle all country dependence by means of easy to understand configuration files. This makes the application more difficult to develop initially, but much easier to adapt to new countries or regions.

  In the current implementation, just one single 10-line method (used for displaying a building's address) is country dependent.

- Due to the use of the hybrid data structure described above, the application doesn't intrinsically "know" which input fields are necessary, it only learns much of this information from configuration files. Combined with the fact that the logbook data encompasses many hundreds of fields, this requires a fundamentally different approach to creating user interface forms than the traditional way.

- The hybrid data structure contains one-to-many relationships, where the entities on the many-side can either be fully country specific (and hence stored in JSON fields) or only partially country specific and stored in their own tables. This fundamental difference must be entirely transparent (i.e. invisible) to the user.

At the present time, the hybrid data structure approach and the configuration-file driven forms have been fully implemented and tested, and the challenges mentioned above have largely been overcome.

The configuration file concept that has been implemented has several advantages:

- The files are in a human readable format and are easy to understand and write.

  As an illustration, the configuration for pilot county Portugal (only for module A, General and administrative information) is shown in appendix A.

- It is flexible with respect to internationalisation. Using only a single configuration file per country or region is supported, but not recommended, since it means that string literals (field labels, labels for

options in dropdown controls, titles, etc.) have to be present in the configuration file itself. This would make translating to application to another language difficult.

Instead, it is recommended to split the configuration files into two parts: the actual configuration file with all the technical field information, and a set of “translation” files that only contain the string literals in a single language. This split-up is shown in appendix A, although only one translation file (for English) is shown.

ANNEX A – INFORMATION ON CONFIGURATION FILES

As an illustration, the configuration for pilot county Portugal (only for module A, General and administrative information) is shown below. It consists of two files, both in YAML, a human-readable format often used for configuration files.

The first file contains a description of the fields to be used for module A for buildings in Portugal. The second file contains the English-language string literals needed to create a user interface from the first file.

Please note that the first file does contain some string literals (the names of the NUTS III regions in Portugal), as it did not make sense to translate names of regions.

The main configuration file (first file) contains two main sections:

- "true_fields": list of fields actually present in the database (table columns), in the order in which the corresponding fields should be displayed in the form. In between fields, titles can be added.
- "json_fields": every field in the "true_fields" list that is a JSON field is represented in the form by the description given for it under "json_fields". For instance, the "address" field shown in "true_fields" is a JSON field. Looking under "json_fields", we see that (for Portugal) the address contains fields for latitude, longitude, NUTS III code, city, etc., so these fields will be used in the form.

First file (field description):

```
building_state:
  general:
    true_fields:
      title1:  # Building
        type: title
        level: 1
      json_building_id:  # building_id was already in use as foreign key, so we
call the field json_building_id
      address:
      property_id:
      title2:  # Building general features
        type: title
        level: 2
      construction_year:
      total_net_floor_area:
        units: m²
      gross_heated_volume:
        units: m³
      building_general_features_other:
      title3:  # Licenses and plans
        type: title
        level: 2
      licenses_and_plans:
      title4:  # Conservation status
        type: title
        level: 2
      conservation_status:
        type: enum
        options:
          option_1:
          option_2:
          option_3:
```

```
        option_4:
        option_5:
    conservation_status_other:
    title5:  # Building Unit
      type: title
      level: 1
    building_unit_building_general_features:
    building_unit_licenses_and_plans:
    title6:  # Building unit conservation status
      type: title
      level: 2
    building_unit_conservation_status:
      type: enum
      options:
        option_1:
        option_2:
        option_3:
        option_4:
        option_5:
    building_unit_conservation_status_other:
    building_unit_other_information:
    building_owner:
    building_user:
json_fields:
  json_building_id:
    fields:
      national_code:
        type: integer
      inspire_id:
        type: string
      energy_supplier_id:
        type: string
  address:
    fields:
      latitude:
        type: float
        units: DD
      longitude:
        type: float
        units: DD
      nuts_iii:
        type: enum
        options:
          option_1:
            name: Alto Minho
          option_2:
            name: Cávado
          option_3:
            name: Ave
          option_4:
            name: Área Metropolitana do Porto
          option_5:
            name: Alto Tâmega
          option_6:
            name: Tâmega e Sousa
```

```
        option_7:
          name: Douro
        option_8:
          name: Terras de Trás-os-Montes
        option_9:
          name: Algarve
        option_10:
          name: Oeste
        option_11:
          name: Região de Aveiro
        option_12:
          name: Região de Coimbra
        option_13:
          name: Região de Leiria
        option_14:
          name: Viseu Dão-Lafões
        option_15:
          name: Beira Baixa
        option_16:
          name: Médio Tejo
        option_17:
          name: Beiras e Serra da Estrela
        option_18:
          name: Área Metropolitana de Lisboa
        option_19:
          name: Alentejo Litoral
        option_20:
          name: Baixo Alentejo
        option_21:
          name: Lezíria do Tejo
        option_22:
          name: Alto Alentejo
        option_23:
          name: Alentejo Central
        option_24:
          name: Região Autónoma dos Açores
        option_25:
          name: Região Autónoma da Madeira
    city:
      type: string
    zip_code:
      type: integer
    street_name:
      type: string
    house_number:
      type: string
    floor_number:
      type: integer
property_id:
  fields:
    legal_registration:
      type: string
    fiscal_registration:
      type: string
building_general_features_other:
```

```
  fields:
    building_use:
      type: enum
      options:
        option_1:
        option_2:
        option_3:
    ownership_type:
      type: enum
      options:
        option_1:
        option_2:
        option_3:
    renovation_year:
      type: integer
    roof_height:
      type: float
      units: m
    altitude:
      type: float
      units: m
    number_of_floors:
      type: integer
    number_of_units_dwellings:
      type: integer
    main_orientation:
      type: enum
      options:
        option_1:
        option_2:
        option_3:
        option_4:
        option_5:
        option_6:
        option_7:
        option_8:
    common_areas_net_floor_area:
      type: float
      units: m²
    photo:
      type: file
licenses_and_plans:
  fields:
    urban_licenses:
      array:
        fields:
          description:
            type: string
          file:
            type: file
conservation_status_other:
  fields:
    evaluation_date:
      type: date
building_unit_building_general_features:
```

```
fields:
  building_unit_total_net_floor_area:
    type: float
    units: m²
  building_unit_conditioned_floor_area:
    type: float
    units: m²
  building_unit_rooms:
    array:
      fields:
        type_of_room:
          type: enum
          options:
            option_1:
            option_2:
            option_3:
            option_4:
            option_5:
            option_6:
        net_floor_area:
          type: float
          units: m²
        depth_regarding_main_facade:
          type: float
          units: m
        floor_to_ceiling_height:
          type: float
          units: m
        volume:
          type: float
          units: m³
  building_unit_floor_to_ceiling_height:
    type: float
    units: m
  building_unit_floor_position:
    type: integer
  building_unit_building_inertia:
    type: enum
    options:
      option_1:
      option_2:
      option_3:
  building_unit_main_orientation:
    type: enum
    options:
      option_1:
      option_2:
      option_3:
      option_4:
      option_5:
      option_6:
      option_7:
      option_8:
  building_unit_number_of_occupants:
    type: integer
```

```
    building_unit_number_of_parking_places:
      type: integer
    building_unit_brief_description:
      type: string
      text_area: true
    building_unit_photo:
      type: file
building_unit_licenses_and_plans:
  fields:
    design_plans:
      array:
        fields:
          description:
            type: string
          file:
            type: file
building_unit_conservation_status_other:
  fields:
    evaluation_date:
      type: date
building_unit_other_information:
  fields:
    governmental_taxes_and_incentives:
      type: string
      text_area: true
    fincancial_programs:
      type: string
      text_area: true
    real_estate_information:
      type: string
      text_area: true
    energy_and_construction_market:
      type: string
      text_area: true
building_owner:
  fields:
    owner_name:
      type: string
    owner_contact:
      type: string
building_user:
  fields:
    user_name:
      type: string
    user_contact:
      type: string
```

Second file: string literals in English. A similar file can be created for any desired language.

```
en:
  cs_model_forms:
    building_state_form:
      general:
        titles:
          title1: Building
          title2: Building General Features
          title3: Licenses and plans
          title4: Conservation status
          title5: Building Unit
          title6: Building Unit conservation status
        json_building_id:
          label: Building ID
          national_code: National Code
          inspire_id: Inspire ID
          energy_supplier_id: Energy Supplier ID
        address:
          label: Address
          latitude: Latitude
          longitude: Longitude
          nuts_iii:
            label: NUTS III
          city: City
          zip_code: Zip Code
          street_name: Street Name
          house_number: House Number
          floor_number: Floor Number
        property_id:
          label: Property ID
          legal_registration: Legal registration
          fiscal_registration: Fiscal registration
        construction_year: Construction year
        total_net_floor_area: Total net floor area
        gross_heated_volume: Gross heated volume
        building_general_features_other:
          building_use:
            label: Building use
            options:
              option_1: Residential
              option_2: Non-Residential
              option_3: Mixed
          ownership_type:
            label: Ownership type
            options:
              option_1: Private
              option_2: Local administration
              option_3: Central administration
          renovation_year: Renovation year
          roof_height: Roof height
          altitude: Altitude
          number_of_floors: Number of floors
          number_of_units_dwellings: Number of units/dwellings
          main_orientation:
            label: Main Orientations
```

```
    options: &orientations
      option_1: North
      option_2: Northeast
      option_3: East
      option_4: Southeast
      option_5: South
      option_6: Southwest
      option_7: West
      option_8: Northwest
  common_areas_net_floor_area: Common areas net floor area
  photo: Photo
licenses_and_plans:
  urban_licenses:
    label: Urban licenses
    array:
      label: Urban license
      new: New urban license
    description: Description
    file: File
conservation_status:
  label: Conservation status
  options: &conservation_statuses
    option_1: Bad
    option_2: Not so good
    option_3: Reasonable
    option_4: Good
    option_5: Excellent
conservation_status_other:
  evaluation_date:
    label: Evaluation date
building_unit_building_general_features:
  label: Building unit general features
  building_unit_total_net_floor_area: Total net floor area
  building_unit_conditioned_floor_area: Conditioned floor area
  building_unit_rooms:
    label: Rooms
    array:
      label: Room
      new: New room
    type_of_room:
      label: Type of room
      options:
        option_1: Common area
        option_2: Bedroom
        option_3: Kitchen
        option_4: Living room
        option_5: Bathroom
        option_6: Corridor/storageroom
    net_floor_area: Net floor area
    depth_regarding_main_facade: Depth regarding main facade
    floor_to_ceiling_height: Floor to ceiling height
    volume: Volume
  building_unit_floor_to_ceiling_height: Floor to ceiling height
  building_unit_floor_position: Floor position
  building_unit_building_inertia:
```

```
    label: Building inertia
    options:
      option_1: Low
      option_2: Medium
      option_3: High
  building_unit_main_orientation:
    label: Main orientation
    options:
      <<: *orientations
  building_unit_number_of_occupants: Number of occupants
  building_unit_number_of_parking_places: Number of parking places
  building_unit_brief_description: Brief description
  building_unit_photo: Photo
building_unit_licenses_and_plans:
  label: Building unit licenses and plans
  design_plans:
    label: Design plans
    array:
      label: Design plan
      new: New design plan
    description: Description
    file: File
building_unit_conservation_status:
  label: Building unit conservation status
  options:
    <<: *conservation_statuses
building_unit_conservation_status_other:
  evaluation_date:
    label: Evaluation date
building_unit_other_information:
  governmental_taxes_and_incentives: Governmental taxes and incentives
  fincancial_programs: Financial programs
  real_estate_information: Real estate information
  energy_and_construction_market: Energy and construction market
building_owner:
  label: Building owner
  owner_name: Name
  owner_contact: Contact
building_user:
  label: Building user
  user_name: Name
  user_contact: Contact
```

ANNEX B – INFORMATION ON CODE PROVIDED

The iBRoad Logbook is a Rails web application. In order to run it in development mode, you'll need to do the following:

- Install the ruby language, preferably using a ruby version manager such as `rvm` or `rbenv`. This application uses Ruby 2.4.3.
- Possibly, create a gemset specifically for this application (refer to the documentation of your ruby version manager)
- Install the application dependencies by running `bundle install` in the application's root directory (the same directory that contains the respective readme file).
- Install and configure PostgreSQL. Any version higher than 9.4 should work.
- Create a database in PostgreSQL. Also create a user and give the user full access to the database.
- Edit the file `config/database.yml` and make the following changes:
  - Enter the user's username and password in the `user` and `password` field under `default`.
  - Enter the name of the database in the `database` under `development` (or under `test` or production if you want to use these environments).
- Run `rake db:seed`
- Run `rails s` to start the server.
- Point your browser to `http://localhost:9292` to start using the application. You can log in with username `a@b.com` and password `letmein`.

Please note that this application expects to run under a linux-like environment (i.e. linux, macOS, OpenBSD, etc.). It is not expected to run without issues on Windows.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 754045