ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Lab Project

Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Smart Heating System in Residential Areas

by Michael Spiegel
and Samuel Schmid

Spring 2015

ETH student ID:	10-915-312 (Michael Spiegel)
	10-919-991 (Samuel Schmid)
E-mail address:	spiegelm@student.ethz.ch
	schmisam@student.ethz.ch
Supervisors:	Wilhelm Kleiminger MEng
	Prof. Dr. Friedemann Mattern
Date of submission:	January 5, 2012

Abstract

English abstract.

Zusammenfassung

Zusammenfassung auf Deutsch.

Contents

1 Introduction

Motivate your problem and outline the contributions of the thesis. [?]

2 Related Work

Put related work here [**?**].

Nico Eigenmann: Prototype infrastructure in university offices [**?**].

CoAP, REST

wenig wissenschaftlich: Django, Android

3 Requirements Elicitation

3.1 Functional Requirements

Functional requirements are stated using use cases in the style of Martin Fowler:
`http://en.wikipedia.org/wiki/Use_case#Martin_Fowler`
`http://ontolog.cim3.net/cgi-bin/wiki.pl?UseCasesMartinFowlerSimpleTextExample`

3.1.1 Register a Residence

1. User opens the app and opens the registration screen
2. User scans RFID tag
3. System checks if RFID is not yet registered
4. System registers residence with scanned RFID
5. System shows empty home screen with no rooms

Alternative RFID already registered

- At step 3, system fails to verify that RFID is not yet registered
- If RFID is associated to a residence
  - System shows the home screen of the registered residence
- If RFID is associated to a thermostat
  - System shows ???

4 System Overview

The system consists of two major parts: The infrastructure gathering and controlling temperature and organisational data, and the mobile applicaiton offering a user interface to view and control the temperature data and residential information. See Figure 4.1 for a graphical overview.

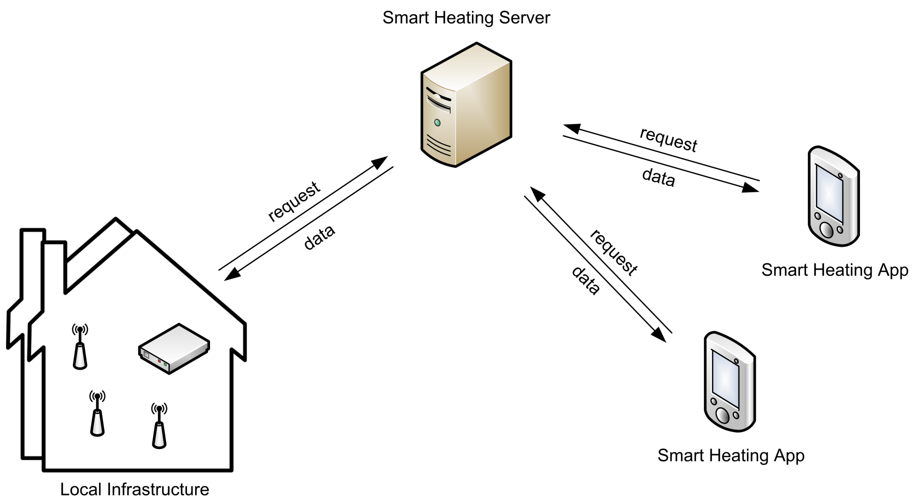

Figure 4.1: System Overview

The infrastructure consists of a local residential part and a server part. Within the residence a small computer system is installed which serves as a communication gateway to the distributed low-power thermostats. Whereas the server is the central entity to collect and store the accumulated sensor data and organizational data. Both parts are explained in Chapters 5 Infrastructure and 6 Mobile App.

The mobile application part . . .

SAMUEL: Mobile App text

4.1 Architecture

The Smart Heating system is implemented using a server-client pattern. The Smart Heating server provides an API which is used by the local communication gateway and the mobile application.

4.2 Models

Throughout the project there is a shared system model describing the underlying entities. The system model depicts the physical objects and places that are used for data collection or organizational purposes. The following paragraphs describe the applied models and their associated design decisions.

Residence The fundamental unit of each deployment is the residence. Each residence corresponds to an installed local communication gateway. Further details are described in Section 5.2

User Each residence can contain multiple users. A user is associated to exactly one residence and is identified by his smart phone serial number. This design decision simplifies the system design and especially the user authentication. It also prevents a user from using the same identity when accessing the system from multiple smart phones.

Room and Thermostat A residence is divided in rooms where each room can contain several thermostats. A room is a simple organizational approach to group multiple thermostats into a single unit.

Heating Table Each thermostat has an associated temperature schedule called heating table. The heating table is responsible for mapping each day and time in a week to a target temperature. The heating table is a periodic schedule repeating each week. This design decision was chosen for infrastructure simplicity as well as to reduce usability complexity.

Meta Entries Meta entries persist time depending meta information about thermostats. A meta entry consists of the received signal strength, up-time, battery level and an associated timestamp. This data can be used to identify issues regarding the thermostat devices such as wireless connection problems or drained batteries.

5 Infrastructure

The infrastructure implements a server-client design pattern and is divided into two parts: The *server* part and the *local* part. The server is an independent unit providing a public API to its clients. The local part is a client using the provided API to retrieve configuration from the server and populate it with data gathered from the deployed residential infrastructure. Both parts will be explained in the following sections.

5.1 Server Infrastructure

The *Smart Heating server* is the central storage and communication center of this project. It persists data collected by the local deployment as also organizational information and heating schedules provided by the user via the Mobile App.

5.1.1 Design Goals

- *Modularity* for independent, interchangeable components for improved maintainability.
- *Extensibility* to easily add new resources and relationships.
- *Usability* to allow developers simple inspection of the API structure and modification of resources.
- *Testability* for good and comprehensible tests and high software quality.

5.1.2 Platform and Frameworks

The server is implemented with Python, a general-purpose, multi-paradigm programming language[1]. Python was chosen as it is suitable for developing a solid web server as well as the embedded hardware used for the communication gateway in the local deployment. See Section 5.2 for more explanation. Django is a popular, open-source web application framework[2] based on a model-view-controller (MVC) pattern facilitating the development of complex, database-driven web applications. The Django REST Framework[3] extends Django to support the design of RESTful[4] Web APIs.

5.1.3 RESTful API

The Smart Heating Server provides a RESTful API to access a persistent storage providing the basic CRUD operations: Create, Read, Update and Delete. Representational State Transfer (REST) is a programming paradigm used for machine-to-machine communication in distributed systems. RESTful web services use HTTP as their preferred communication protocol.

REST is about resources. Each resource is identified by a uniform resource identifier (URI) and is accessible via the request methods defined in HTTP. We differentiate two major types of URIs: Resource collections and resource representations. A resource representation is a view of its resource's state and is encoded in a transferable format. This project uses the simple JavaScript Object Notation (JSON) format to represent resources. Resource collections contain lists of representations of the same type of a resource.

Design Decisions

1. The underlying model hierarchy is represented via nested URLs.
2. The resource identifier is the first field in each resource representation.
3. Within resource representations, collections are referenced via URL. Resources are referenced by including their representation.

[1] `https://www.python.org/`
[2] `https://www.djangoproject.com/`
[3] `http://www.django-rest-framework.org/`
[4] Note: REST is a programming paradigm where as RESTful is used to describe a web application implementing such a paradigm.

4. URL fields are identified by the name `url` or the suffix `_url`. Each resource representation contains its own URL in the field `url`.

See Listing 5.1 for an example of a resource representation.

```json
{
  "id": 2,
  "url": "http://server/residence/04891BB9232584/room/2/",
  "name": "Dining Room",
  "residence": {
    "rfid": "04891BB9232584",
    "url": "http://server/residence/04891BB9232584/",
    "rooms_url": "http://server/residence/04891BB9232584/room/",
    "users_url": "http://server/residence/04891BB9232584/user/"
  },
  "thermostats_url": "http://server/residence/04891BB9232584/room/2/thermostat/"
}
```

Listing 5.1: Example representation of the Room resource at `http://server/residence/04891BB9232584/room/2/`. The `url` field determines the URL of the represented resource. Within the `residence` field the representation of the associated Residence resource is nested. The included Residence representation has its own `url` field. Collections of the Residence's Rooms and Users are not nested but referenced via URL to limit the response size.

Browsable API The Django REST Framework offers the ability to dynamically generate a browsable interface accessible via web browser. It includes a formatted output of the JSON response, offers forms to add new resources and edit or delete existing resources, and displays URLs as clickable hyperlinks. This interface is an additional HTML output format enabling developers to easily interact with the API without the need of external tools. The Browsable Web API is designed for readability whereas the API renders its data as compressed JSON to reduce transmission bandwidth.

MICHAEL: "bandwith" - Can't think of the correct word..

See Figure 5.1 for an example.

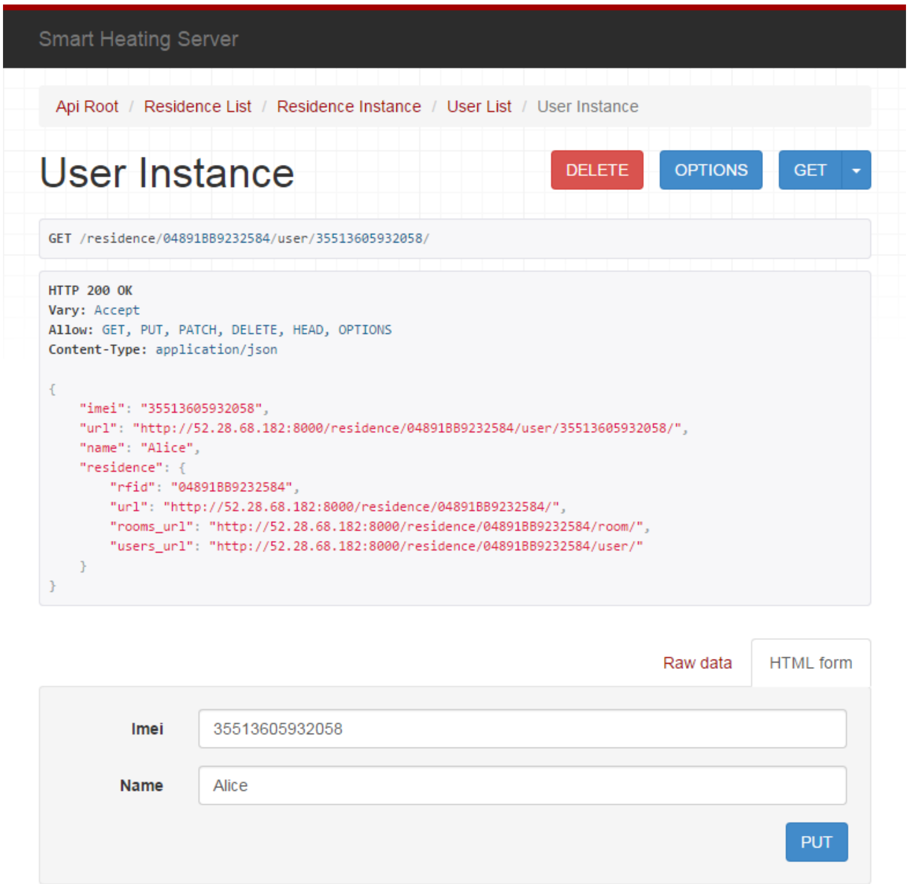

Figure 5.1: Screenshot of the Browsable Web API showing an instance of a user resource. The interface allows the developer to easily edit or delete the user. Navigating the residence or the associated room or user collections can be done by clicking on the URLs of the respective resources.

5.1.4 Program Architecture and Implementation

The server implementation is built on Django following a variation of the Model-View-Controller (MVC) architectural pattern. We assume the reader to be familiar with the common MVC pattern[5].

All the code required to setup the Smart Heating server is provided in the GitHub repository located at `https://github.com/spiegelm/smart-heating-server`. We highly encourage the reader to visit the project on GitHub. The main page of the repository provides a short explanation and also contains a hyperlinked image indicating the status of the latest build run on the continuous integration service as explained later in Section 5.1.5. The repository is designed to contain all code and

[5]We refer the interested reader to `https://en.wikipedia.org/wiki/Model-view-controller`

dependency information necessary to automatically install and run the application on a server.

The following sections describe the relevant application components in detail.

5.1.4.1 Models

Models structure the underlying data and provide operations for manipulating it. In Django models contain the business logic and are also responsible for validating user input and providing appropriate error messages. The `smart_heating.models` namespace contains all project models. Each model extends the abstract `Model` class, providing a general method used to get an objects representation for debugging purposes as well as the abstract `get_recursive_pks` method. This method is used to generate hierarchical URLs and will be further described in Section 5.1.4.2

Thermostats are organized within rooms and belong to a residence. RaspberryDevice and ThermostatDevice represent the deployed physical devices. Both are used as a global configuration of all devices that are planned to be eventually installed in a residence. This configuration associates a device's MAC address to the number of the radio-frequency identification (RFID) tag attached on the device. This is required to relate these two identifiers, as the hardware cannot read the RFID tag number from the device casing and the mobile application cannot access the internal MAC address used by the devices. Storing this mapping on the server benefits the deployment of the local devices as every device can use the same code and no configuration of the devices is required. See Figure 5.2 for a graphical representation of the used models. The most important models will be described in the following paragraphs.

Residence is identified by the RFID tag number on the deployed local communication gateway. A residence combines users and rooms with their associated thermostats and data into a single encapsulated unit.

User is identified by the smart phone's serial number[6] and can be registered to at most one residence.

[6] The International Mobile Equipment Identity (IMEI) is a 15-digit serial number associated to each GSM cell phone.

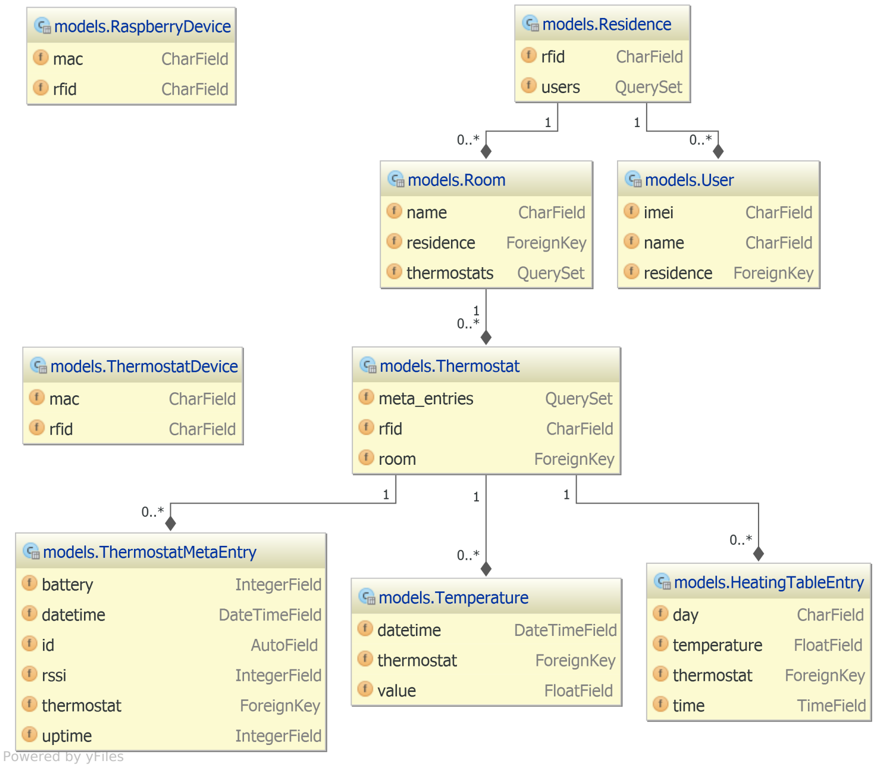

Figure 5.2: UML class diagram of the used models.

Room groups thermostats into an organizational unit. Each residence can contain multiple rooms.

Thermostat is identified by the number of the attached RFID tag. It is the parent of the models used for the historic temperature and thermostat meta data and the heating schedule.

Temperature represents a single temperature measurement perceived by a thermostat at a specific date and time.

ThermostatMetaEntry is used to combine meta data that is available by the digital thermostat such as battery level, system uptime and received signal strength indication (RSSI).

HeatingTableEntry is used to compose the heating schedule for a thermostat. An entry determines the target temperature that applies starting from the given day and time until the next heating table entry occurs.

5.1.4.2 Serializers

Serializers are responsible for translating models into a resource representation and vice versa. This project uses JSON to represent resources. Each serializer has an associated model and determines which model fields should be included into the resource representation. Additional fields can be added to the representation or individual field representations can be overwritten to offer customized output.

A commonly used customization is the replacement of the default `HyperlinkedIdentityField` with `HierarchicalHyperlinkedIdentityField`. This custom serializer field generates URLs according to the hierarchical schema applied in this project. For example the URL for a room would be `http://server.com/residence/041FB2B9232580/room/5/`. To assemble this URL the identifier of the room and its parent are required. The adapted field extends the default field and provides all identifiers of the hierarchy required to generate the URL by using the hierarchical base class `smart_heating.models.Model`.

Nested resources are implemented using nested serializers. This way existing serializers can be reused to include their resource representation into another representation. Refer to line 3 in Listing 5.2 for an example usage of a nested serializer.

```
1 class RoomSerializer(serializers.HyperlinkedModelSerializer):
2   url = relations.HierarchicalHyperlinkedIdentityField(view_name='
      room-detail', read_only=True)
3   residence = ResidenceSerializer(read_only=True)
4   thermostats_url = relations.HierarchicalHyperlinkedIdentityField(
      source='thermostats', view_name='thermostat-list',
5   read_only=True)
6
7   class Meta:
8     model = Room
9     fields = ('id', 'url', 'name', 'residence', 'thermostats_url')
```

Listing 5.2: The `RoomSerializer` class as an example of the usage of the `HierarchicalHyperlinkedIdentityField` and nested serializers.

5.1.4.3 Views

A view is a class which processes requests and returns a response. Django defines the view as the place that not only defines how but also which data is presented. This slightly differs from the traditional MVC architectural pattern but will not be explained in detail here[7]. Django includes generic view classes and mixins to provide common functionality and to facilitate code reuse. The Django REST Framework further abstracts views by supplying `ViewSets`. These `ViewSets` facilitate the unified handling of different HTTP methods within a single class for each public resource.

The used `ModelViewSet` is such a class serving as a base for individual view sets representing a resource. In the very simplest case only the definition of a database query and an appropriate serializer class is required. The `ResidenceViewSet` is a good example of a fully functional resource implementation supporting all CRUD methods while only requiring three lines of code, as shown in Listing 5.3.

```
class ResidenceViewSet(viewsets.ModelViewSet):
  queryset = Residence.objects.all()
  serializer_class = ResidenceSerializer
```

Listing 5.3: The `ResidenceViewSet` class.

Pagination is the practice of partitioning lists into smaller pieces. In this project pagination is used for expectedly large resource collections such as temperatures or meta entries. The applied pagination style depends on two parameters: `limit` and `offset`. Limit determines the page size, i.e. the maximum number of items that are displayed simultaneously. Offset determines the number of the starting item. The custom pagination class `pagination.BasePagination` implements the design decision 4 to suffix resource field names representing links.

5.1.4.4 Routes

URLs are defined explicitly in the file `smart-heating/urls.py`. For each request the responsible view is determined by matching the requested URL against a list of regular expressions. This technique allows a flexible and clean URL schema as required for our hierarchical resources.

[7]For more details we refer the reader to `https://docs.djangoproject.com/en/1.8/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names`

5.1.5 Automated Testing

Automated software testing is an important part of this project part. Django and also the Django REST Framework facilitate automatic testing by providing base classes and tools helping to create and execute tests.

The tests are designed to infer the functionality and behavior of the server application. An executed test run shows at any particular time which functional requirements are fulfilled and which are not. Look at Listing 5.4 for an example. These dynamically generated test reports nicely complement traditional documentation and the tests also provide small informational code examples that are shown to work. During application development we intensively used test driven development (TDD) practices to increase productivity and achieve high software quality. Automatic software testing allows to formulate the requirements of a computer program such that they can be automatically checked already before and during application development.

```
$ python manage.py test -v 2
  [...]
  smart_heating.tests.test_api.ViewRootTestCase
    test_root_contains_residence_url ... ok
  smart_heating.tests.test_api.HeatingTableTestCase
    test_create_heating_table_entry ... ok
    test_heating_table_entries_are_ordered_by_date_and_time ... ok
    test_heating_table_entry_representation ... ok
    [...]
  smart_heating.tests.test_api.ViewUserTestCase
    test_create_user ... ok
    test_get_non_existent_imei_is_404 ... ok
    test_get_user_of_unrelated_residence_is_404 ... ok
    test_user_collection_contains_user_representations ... ok
    test_user_representation_contains_imei_and_name ... ok
    [...]
--------------------------------------------------------------------
Ran 58 tests in 0.765s
OK
```

Listing 5.4: Excerpt of the test output documenting the application behavior

Furthermore the usage of an Continuous Integration service like Travis CI[8] ensures that individual development branches are consistently tested and don't break the

[8] `https://travis-ci.org/`

main development line upon branch integration. Additionally this convenient services ensures the periodic execution of all tests and logging of the test results.

5.2 Local Deployment

The local deployments consists of the residential communication gateway and the deployed digital thermostats with their wireless adapters. The communication gateway collects the data read from the thermostats and sends it to the remote web server. The digital thermostats are programmable and allow us to modify their behavior by flashing custom firmware. This project uses the work of a previous master thesis as a basis to build upon [?]. The primary focus is to improve the basic functionality of the communication gateway and create an unified but loosely coupled infrastructure by using the RESTful API provided by the server. See also Figure 5.3 for an overview of the local deployment.

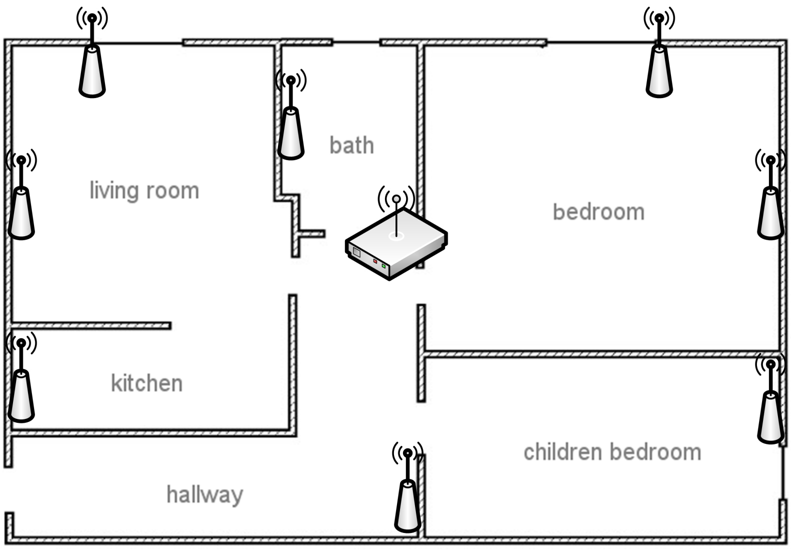

Figure 5.3: Example of a residence layout depicting a possible deployment. The local communication gateway is installed in the hallway, connected to the internet and has wireless connections to the deployed thermostats represented as antennas. Source of the original image: `http://www.haus-topplicht.de/wp-content/uploads/2013/12/planwohnung2.jpg`

5.2.1 Existing Infrastructure

This project builds upon work previously done by Nico Eigenmann as his master thesis [?]. Part of his work consisted of extending the hardware and software of the digital thermostat HR-20[9], as depicted in Figure 5.4, to offer wireless access via 6LoWPAN[10]. Further a border router translates the 6LoWPAN messages into IPv6 packets and vice versa. This border router is connected per Universal Serial Bus (USB) port to a computer and communicates via Serial Line Internet Protocol (SLIP). The computer redirects the received packets into the connected Local Area Network (LAN) and also routes messages addressed to a sensor in the 6LoWPAN network via the border router.

Figure 5.4: Honeywell Rondostat HR-20 programmable thermostat. Source: `https://piontecsmumble.files.wordpress.com/2013/02/hr20.jpg`

5.2.2 Design Goals

Several design goals are determined in order to evaluate the system design and implementation.

- *Performance* for low computational effort and power consumption on embedded systems.
- *Reliability* for a high probability that the system operates as expected.
- *Robustness* to let the system behave reasonably in presence of failures, especially connection problems.
- *Interoperability* to cooperate with other distributed systems.

[9] `http://www.homexpertbyhoneywell.com/en-DE/Products/rondostat/Pages/HR-20.aspx`
[10] 6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Network

5.2.3 Software Platform and Frameworks

This sub section lists and describes the software components used on the local communication gateway.

tunslip6 is a tool to route IPv6 packets between the host and a border router via serial line. It is used to create a tunnel interface an the host system which acts as a regular network interface.

Python is a multi-paradigm programming language and is suitable for desktop as well as for embedded systems. This allows us to use a single programming language for the implemented software parts in the whole infrastructure on the local and server side. Further Python includes packages to facilitate asynchronous input and output which is required to concurrently query multiple network resources.

Constrained Application Protocol (CoAP) [?] is a application protocol created for low power devices and sensors that are heavily restricted in terms of computing power, memory size and power consumption. Unlike the Hypertext Transfer Protocol (HTTP) commonly used in desktop and mobile systems, CoAP is based on minimalistic network protocols to reduce overhead and computational requirements.

aiocoap is a third-party package for Python implementing the CoAP protocol.

requests is a library written in Python providing a simple and well designed interface to send HTTP requests. This is used to communicate with the API provided by the Smart Heating Server.

5.2.4 Implementation

The implementation work on the local infrastructure focuses on the local computer acting as a local communication gateway, this two terms are further used interchangeably. We use the Raspberry Pi 2 Model B[11] as our local communication gateway which is responsible for two major tasks.

[11] `https://www.raspberrypi.org/products/raspberry-pi-2-model-b/`

First, it needs to periodically query the temperatures and other meta data from the digital thermostats and store the data locally. Additionally the heating scheduling algorithm determines the current target temperature and applies the value to the thermostats. Second, the local computer consistently synchronizes the local data storage with the remote Smart Heating Server. The new data gathered from the thermostats is therefore sent to the server and the latest configuration such as the heating schedule is downloaded.

The local computer works as a proxy server and enables the local deployment to operate independently from the connection to the remote server. This way the last downloaded heating schedule is kept and operated until the server connection is reestablished.

All the code, binaries and instructions required to setup the Raspberry Pi 2 are provided in the Git repository located at `https://github.com/spiegelm/smart-heating-local`. We highly encourage the reader to visit the project on GitHub. The repository is designed to be self contained. This means in order to setup a local communication gateway only the hardware, a copy of the repository and an internet connection are needed. A step-by-step instruction to setup the software on a Raspberry Pi 2 is included in the `README.md` file.

5.2.4.1 Border Router Connection

Upon connection of the border router at an USB port the corresponding rule in `/etc/udev/rules.d/90-local.rules` is executed. The rule executes the file `bin/start_tunslip.sh` in the code repository to establish a tunnel to the border router using tunslip6. The border router runs a web server providing information about connected thermostat devices. The log file `/var/log/tunslip6` generated by tunslip6 shows the associated web server address:

```
Server IPv6 addresses:
 fdfd::212:7400:115e:a9e5
 fe80::212:7400:115e:a9e5
```

The address with the prefix `fdfd:` is the one we want. Requesting the determined URI `http://[fdfd::212:7400:115e:a9e5]` returns an HTML table containing all discovered devices:

```
<html><head><title>ContikiRPL</title></head><body>
Neighbors<pre>fe80::221:2eff:ff00:228b</pre>
Routes<pre>fdfd::221:2eff:ff00:228b/128 (via fe80::221:2eff:ff00
   :228b) 16710893s</pre>
</body></html>
```

Each route corresponds to a CoAP URI. For example the URI `coap://[fdfd::221:2eff:ff00:228b/128]/sensors/temperature` could be queried by a CoAP client to determine the currently measured temperature.

Please note: The discovered devices are not necessarily thermostats registered to the local communication gateway. The linked thermostats are therefore retrieved from the Smart Heating Server to avoid unnecessary querying of unrelated devices.

5.2.4.2 Executable Scripts

The `smart-heating-local` repository contains two python scripts in the repository root: `thermostat_sync.py` and `server_sync.py`. Each script corresponds to one of the two tasks described in Section 5.2.4. The first scripts is responsible for fetching the temperature and other measurements from the thermostats to the local storage and applying the latest heating schedule. The latter script sends the measurements that where not yet delivered to the server and retrieves the most actual heating schedule. Both scripts are independent from each other and use only the local storage to share data as explained in Section 5.2.4.3 below.

Scheduling The periodic execution of the both scripts is scheduled by the *cron* service which is provided by the deployed operating system *Raspbian* and also by most Linux operating systems. The `crontab` program is used for defining the schedule.

5.2.4.3 Local Storage

We combine SQLite and dbm for the local storage. *SQLite* is a lightweight relational database engine implementing most of the SQL standard and is often used on embedded systems. *dbm* is a family of simple database engines allowing to store pairs of a unique key and a value. The two databases are used for different purposes. The SQLite database persists historic measurement entries that should be held at least until they are transferred to the server. On the other side there is the dbm database holding configurations retrieved from the server. For example the thermostats associated with a local communication gateway and the current heating schedule are stored using dbm as it has no relational schema and is therefore much simpler to use and maintain. Both databases are stored in files within the `data/` directory.

5.2.4.4 Application

All Python files containing application logic are located in the directory `smart_heating_local`. The following paragraphs list the files within this directory and describe their purpose.

config.py contains the `Config` class responsible for storing and retrieving the heating schedule as also the thermostat MAC addresses linked to this Raspberry Pi computer. The configuration is stored using the `shelve` Python package which itself builds on the npm database engine installed on the system.

server.py provides an interface to communicate with the Smart Heating Server. It also determines the URI schema used by the server API.

server_models.py contains the models used by the server. The models correspond to the resources provided the server API and structure them into classes.

server_controller.py provides a layer of abstraction by packaging the most important functionality regarding server communication. It makes use of the files `server.py` and `server_models.py`. The provided methods include downloading the latest heating schedule and the thermostat configuration linked to this local communication gateway as well as storing the data in the local storage using the `Config` class.

local_models.py contains the models handling the temperature measurements and meta data entries which are stored in the local database.

thermostat_controller.py is responsible for querying the thermostats defined in the configuration and saving the data into the SQLite database. The controller also contains the logic regarding CoAP requests and the resource schema used by the digital thermostats.

tests/ is the directory containing the tests that were created during implementation and which assisted the application development. The tests can be started using the command `nosetests -a '!local'` from the repository root. These tests are also periodically executed and logged by the Continuous Integration service Travis CI[12].

logging/ contains the central logging configuration used in the whole local project. Relevant information is logged to the file `logs/smart_heating.log`.

[12] `https://travis-ci.org/`

6 Mobile App

Smart heating systems are on the rise. More and more companies are trying to secure a spot in the market offering a variety of features in their control applications, which are mostly mobile or tablet based or even come with their own device. In this section we discuss the Android application we designed and implemented for users to control our heating system. We focused on keeping things simple because as we were researching some of the already existing systems we realized very quickly that the main issue is usability. In almost all cases the user is presented with an abundance of features and extras. Even though most of them would be very useful and effective, the average user will most likely be overwhelmed. Many user interfaces put functionality first. This often results in cluttered designs. Figure 6.1 shows some of the user interfaces for controlling smart heating systems.

In Section 6.1 we look at some use cases for such control systems and talk about how a general control application would handle them. Afterwards we show our own design of the application for control the smart heating system we are presenting.

Because of the lack of easy to use controlling in already existing systems we chose to focus our design decisions on these aspects. The user is confronted only with a small number of screens which are all visually designed in a way so the user will always immediately know what he is looking at. The different screens are discussed in further detail in Sections 6.2.2 through 6.2.5.

Finally we evaluate our design decisions, reflecting on the use cases to see which ones were covered by our application and which ones were not. Naturally during development we came up with a lot of new ideas for new features and extras for our own application but we decided to stick with the initial design choices to keep it as simple as possible. We will talk more about these ideas in Section 8, where possible future work on this project is listed.

Figure 6.1: Some examples of mobile applications for controlling smart heatings systems. Source: `https://cdn.recombu.com/media/digital/news/legacy/M13058/1397569835_w670_h576.png`

6.1 Use Cases

We analyze different use cases that an average user might run into while using a smart heating system. This way we are able to ensure that our design decision leads to a simple yet effective application which helps the user control the system in an easy way without overcomplicating things. There is a tradeoff that certain functionality is lost because of these decisions but our main focus was to create an application which is easy to use.

1. **Use Case: User wants to install the system**

   Anna has just purchased the smart heating system in the store. After she comes home, she unwraps the components and wants to install the system. The manual tells her to install the mobile app to set up the system. The app guides her through the installation process and allows her to connect the thermostats she manually installed on the radiators to the system.

2. **Use Case: User feels cold**

   Bob is sitting in the living room feeling cold. He starts the mobile app to access the temperature settings for the living room. He sees that the current temperature is at 18 C and that the target temperature is 21 C. He realizes that he has already changed the temperature and notices the app tells him how much longer he has to wait for the room to heat up.

3. **Use Case: User wants to save money**

   Jack is short on money. He wants to save as much as he can to get through the month, so he start the mobile app for his smart heating system. The application tells him that he can save up to 10 dollars this month by reducing the target temperature for his bathroom by five degrees. He does like it warm in the bathroom but another pair of socks will do just fine, he thinks.

4. **Use Case: User wants to add a room to an existing system**

   Claire finally got her husband to agree to install the new heating system in his office as well. He does not like to deal with that tech-stuff, so Claire will take care of this. Using the already installed mobile application on her phone she can easily add another room to the system and copy the existing heating schedules.

5. **Use Case: User wants to add a thermostat to a room in the system**

When John bought his heating system he wanted to first try it out with only one of his radiators. Seeing how well the system is behaving he decides to buy new thermostats for all the radiators in his room. Gladly, adding new thermostats to an already existing system is easy using the mobile application. Soon he will save even more money on gas bills.

6. **Use Case: User feels hot**

   Joe and Mary have finally gotten used to the new heating system. Mary usually feels cold very quickly, so she tends to turn the living room's target temperature up a bit too high in Joe's opinion. But this week, she is out of town and Joe can already feel the sweat running down his back, so he opens the mobile application for the heating system and sets a new target temperature for the living room. He can already hear the heater shutting down and is looking forward to a smaller bill and a more comfortable temperature.

6.2 Implementation

6.2.1 Application Flow

As seen in Figure 6.2 there are four different views in the application. These different views are explained in more detail in the following sections. The Welcome View is special, because that one is only shown when the user opens the app for the first time and has not yet registered his raspberry pi with the server. After registration, the application will always start in the Home View and go from there.

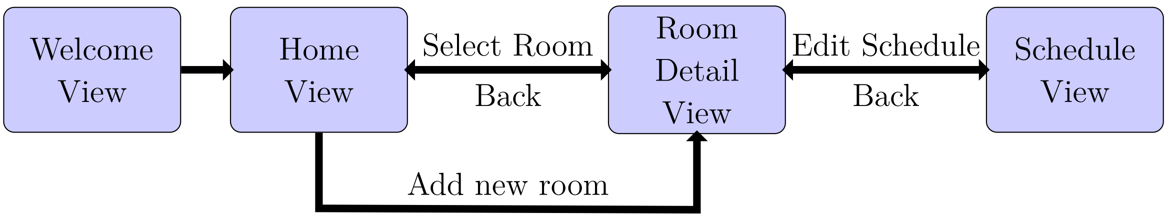

Figure 6.2: The application flow of the mobile app.

6.2.2 Welcome View

In the Welcome View, before being able to use the system the user is prompted to scan his raspberry pi in order to register it with the server. Once the registration is complete the internal database is set up according to the model seen in Figure 6.3. The database is a simple model that keeps track of the rooms and thermostats which the user has added to the system so far, as well as the user's desired heating schedule for each room. To prevent any data inconsistencies, every entry in the table *Room* has a field *server_id*. This field corresponds to the *id* of the room objects on the server.

After the initial setup of the database, the user is confronted with four questions about his daily routine. The questions are:

- When do you usually wake up in the morning?
- When do you usually leave for work in the morning?
- When do you usually get home from work in the evening?
- When do you usually go to bed in the evening?

With the help of these four questions the application is able to set up a default heating schedule which is initially used for all the rooms that are added in the future. It simply takes the answers to the four questions and sets up the heating schedule in the following way:

- If the user is at home, the temperature is set to the default value for when somebody is at home.

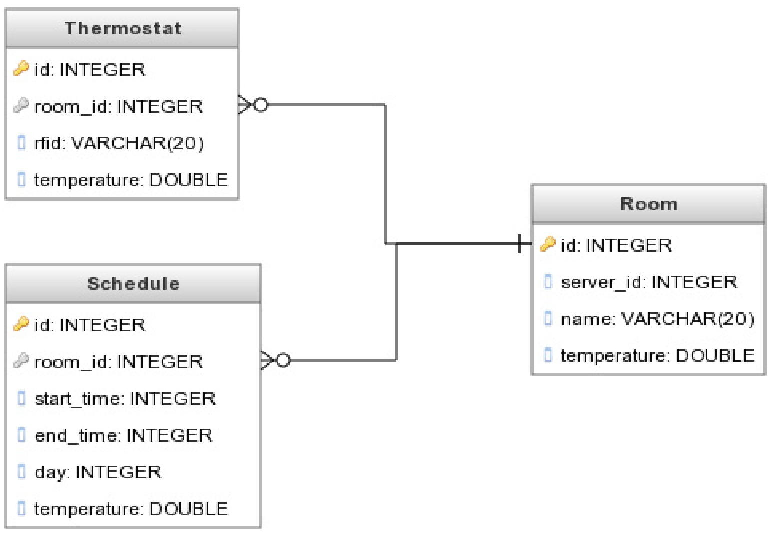

Figure 6.3: The database model used in the mobile application.

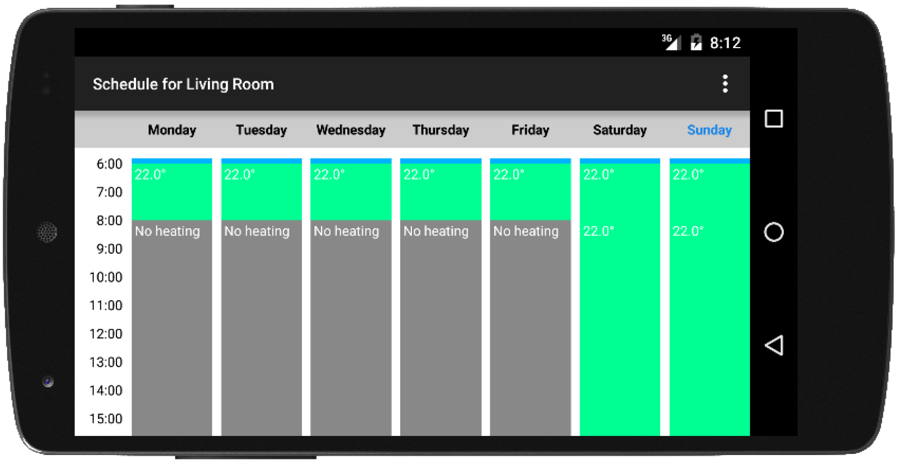

Figure 6.4: A sample default heating schedule after the initial setup of the system.

- If the user is at work, the heating is turned off completely.
- If the user is sleeping, the temperature is set to the default value for the night.
- On weekends, the temperature stays on the default value for being home throughout the whole day.

These default values are initially set to 16 and 22 degrees celsius for being home and sleeping respectively. These values can be changed in the settings menu from the home view (Section 6.2.3).

If desired, the user can also change every heating schedule separately to his own needs. More details about heating schedules can be found in Section 6.2.5.

6.2.3 The Home View

The home view is where the application usually starts after successful registration of the raspberry pi with the server. In the home view the user can see all the rooms he added to the system with the corresponding current temperatures in each room. The colour of the tiles with the room names are also an indicator to how hot the room is in its current state, ranging from blue (cold) to green (normal) to red (hot). Clicking one of the tiles will lead the user to the room detail view described in Section 6.2.4. Next to the current temperature of each room there is an optional flame icon indicating that the room is being heated up at the moment. See Figure 6.5 for an example.

Via the menu in the upper righthand corner, the user can choose to add new rooms or delete existing ones. Adding a new room simply requires a new name for the room to be entered. After the creation of a new room the user is immediately transferred to the room detail view so that he can add new thermostats to the room. For more details about the room detail view see Section 6.2.4.

6.2.4 The Room Detail View

After clicking one of the tiles in the home view the user is presented with more details about the room. He can see a list with all the thermostats, the current temperature in the room indicated with a large thermometer and also a slider next to it for adjusting the desired temperature of the room.

The desired temperature comes from the heating schedule currently active for the room. If the user wants to change the desired temperature he can simply drag the slider to a new position. The heating schedule for the room will be adjusted automatically as well.

If the user wants to change the heating schedule manually he can do so by selecting "View/Edit schedule" from the menu located in the upper righthand corner. This will lead him to the schedule view described further in Section 6.2.5.

picture detail view

6.2.5 The Schedule View

The schedule view shows the heating schedule of the room the user has selected previously in the home view. The application is forced into landscape mode to be able to display all the days of the week. The current day of the week is highlighted for easier usability. The user can scroll up and down to see all the different times of the day.

Adding a new entry is simple: with a single tap anywhere on the schedule the user can add a new entry in the corresponding time slot. He can also adjust the exact times of the new entry after tapping. After providing the desired target temperature for the new entry the user can confirm the data and the new schedule entry is sent to the local database as well as the online server right away.

Removing a heating schedule entry is not possible and should not be necessary. If the user wants to turn off the heating for a certain period he can simply add a new entry and check the "No heating" checkbox. See Figure 6.6 for an example.

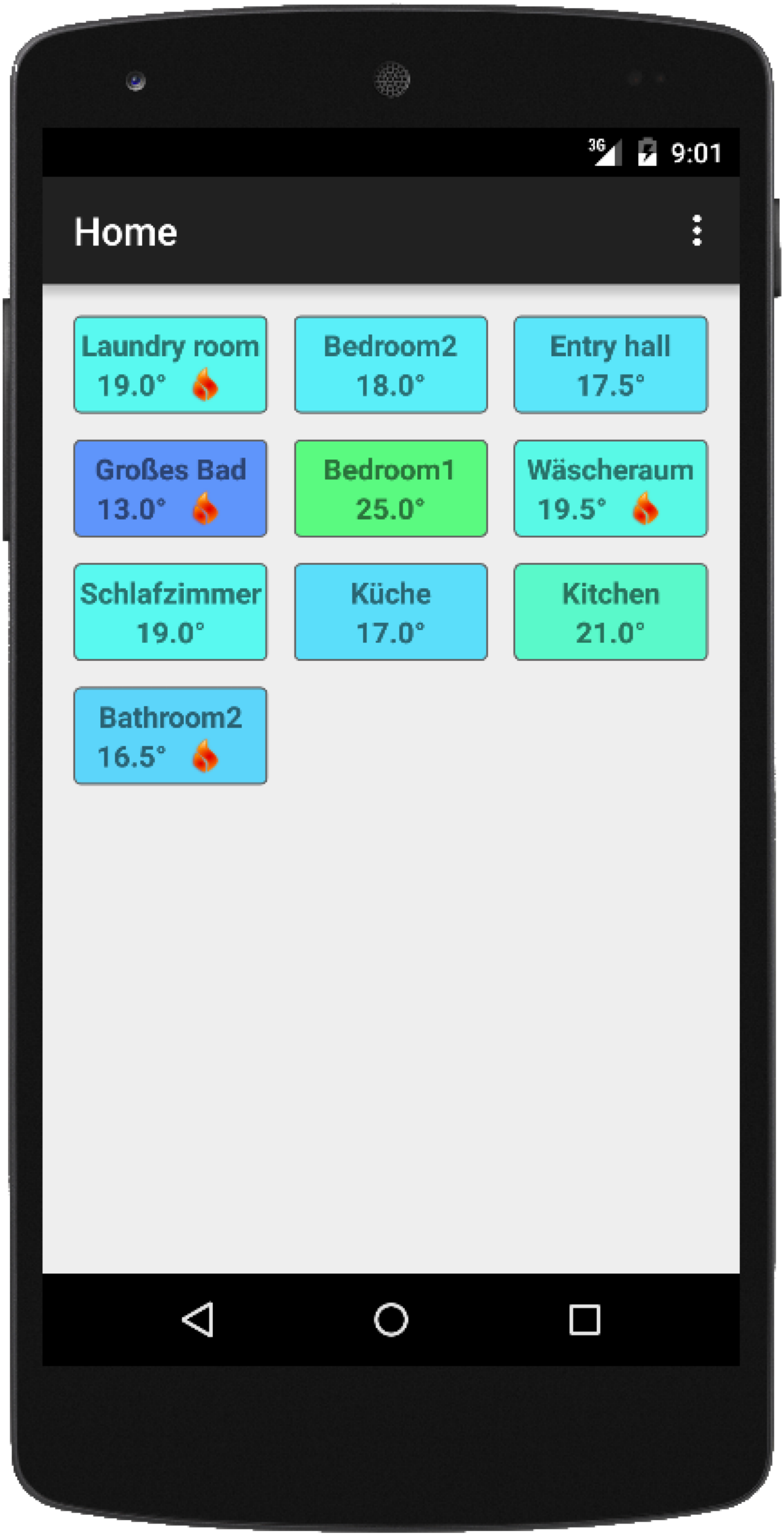

Figure 6.5: A sample home view with some random rooms.

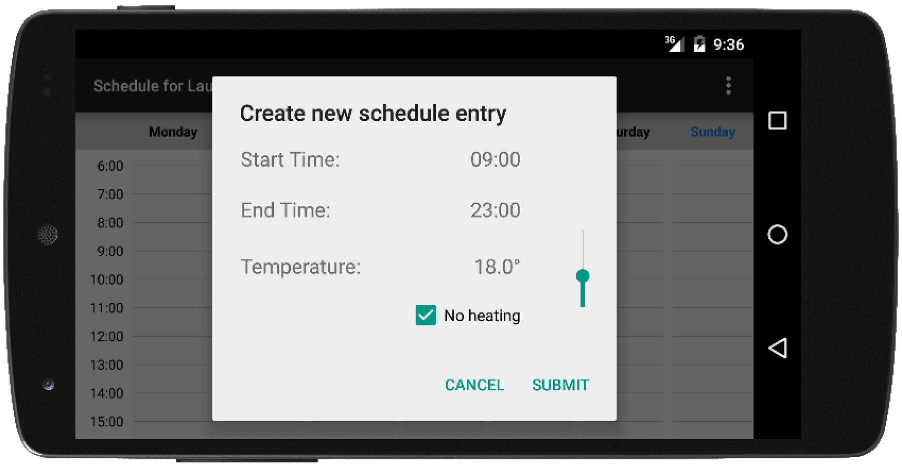

Figure 6.6: The popup for adding a new heating schedule entry.

6.3 Evaluation

6.3.1 Use cases coverage

Our mobile control application covers most of the use cases introduced in Section 6.1 adequately. As shown later in some cases we decided to either leave the feature out for simplicity's sake or because it was out of scope for this lab project.

1. **Use Case: User wants to install the system**

   Covered by the welcome view: The user can easily setup the system using the NFC tags distributed on the raspberry pi and the thermostats with the help of the information displayed on the application.

2. **Use Case: User feels cold**

   Covered by the detail view: The user can adjust the temperature of the room he is currently in using the slider in the detail view.

3. **Use Case: User wants to save money**

   Indirectly covered: The user can choose to set the default temperatures to values lower than usual in order to save money.

4. **Use Case: User wants to add a room to an existing system**

Covered by the home view: The user can simply add a room using the menu in the upper righthand corner.

5. **Use Case: User wants to add a thermostat to a room in the system**

   Covered by the detail view: The user can add a new thermostat to the currently selected room by simply scanning its NFC tag.

6. **Use Case: User feels hot**

   Covered by the detail view: The user can adjust the temperature of the room he is currently in using the slider in the detail view.

6.3.2 User study

We have conducted a user study with some of our fellow students. Here are some of their comments about the usability and the application's design.

Actually do it...

7 Evaluation

Evaluate your algorithm.

Raspi: 27.7 10:30 - 15:45 ausgesteckt 27.7 21:45 - 05:30 development

7.1 Server

- Browsable API. Enables the developer to interactively discover the API structure and behavior

Siehe auch Report Structure.

Design Decisions Django is designed for flat URLs but is still flexible enough to support the design decision of nested resources represented within the URL.

7.2 Local

7.3 Mobile App

8 Future Work

Anything that is interesting – but you did not have the time to pursue – fits into future work.

9 Conclusion

Give a summary on what you did and what the major results are.

A First Appendix

Use appendix for more anything that does not fit into the main document (e.g., implementation details).

Include links to all GitHub repos

Setup instructions for the local infrastructure

insert protocol