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

Cooleren Titel finden?

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:	Dr. Wilhelm Kleiminger
	Prof. Dr. Friedemann Mattern
Date of submission:	TODO

Abstract

In this lab project we present a reliable heating control system for residential users. This project consists of two major parts. On the one hand there is the back end consisting of the local infrastructure monitoring and controlling temperatures and persisting this data on an accessible web server. On the other hand there is the front end as an Android mobile application offering a user friendly interface to configure and control the system. The application is a lightweight distributed software system designed for multiple users in the same residence.

Contents

1 Introduction

The energy consumption is growing world wide and causes increasing costs and environmental damage. In 2013, private households in Switzerland consumed 29% of the country's total energy [2], most of it being used for heating purposes [5]. A lot of different tools in the domain of residential heating control were developed to help reduce the consumed energy for space heating. But most of these tools do either not motivate end users to save energy or are too complex for the end user and thus get discarded in the long-term view. This results in the need of user friendly tools that provide the end user with just enough information to analyze their data but not overwhelm him with unnecessary details.

For this specific area we developed our project, which consists of a reliable infrastructure serving as a solid base, and a well designed distributed mobile application allowing the user to configure and interact with the heating system.

This report is structured as follows: Chapter 2 treats the process of determining the requirements of the infrastructure and applications developed in the scope of this lab project. Chapter 3 gives an overview of the whole system environment and introduces the chosen architectures of our application. Chapter 4 explains the developed infrastructure with the local deployment and the communication server in detail. Chapter 5 describes the implemented mobile application in detail. In Chapter 6 we evaluate the developed system considering the defined requirements and design goals.

2 Requirements Elicitation

SAMUEL

2.1 Functional Requirements

Functional requirements are stated using use cases in the style of Martin Fowler:

Add 2 sentences describing this style and put the links in footnotes or references.

http://en.wikipedia.org/wiki/Use_case#Martin_Fowler
http://ontolog.cim3.net/cgi-bin/wiki.pl?UseCasesMartinFowlerSimpleTextExample

2.1.1 Register a Residence

Add one to two sentences explaining what the user wants to achieve (i.e. user has just bought the system and wants to enable smart heating in her 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 ???

3 System Overview

The system consists of two major parts. First, the back end consisting of the local infrastructure monitoring and controlling temperatures and persisting this data on an accessible web server. Second, the mobile application offering a user friendly interface to configure and interact with the system. See Figure 3.1 for a graphical overview.

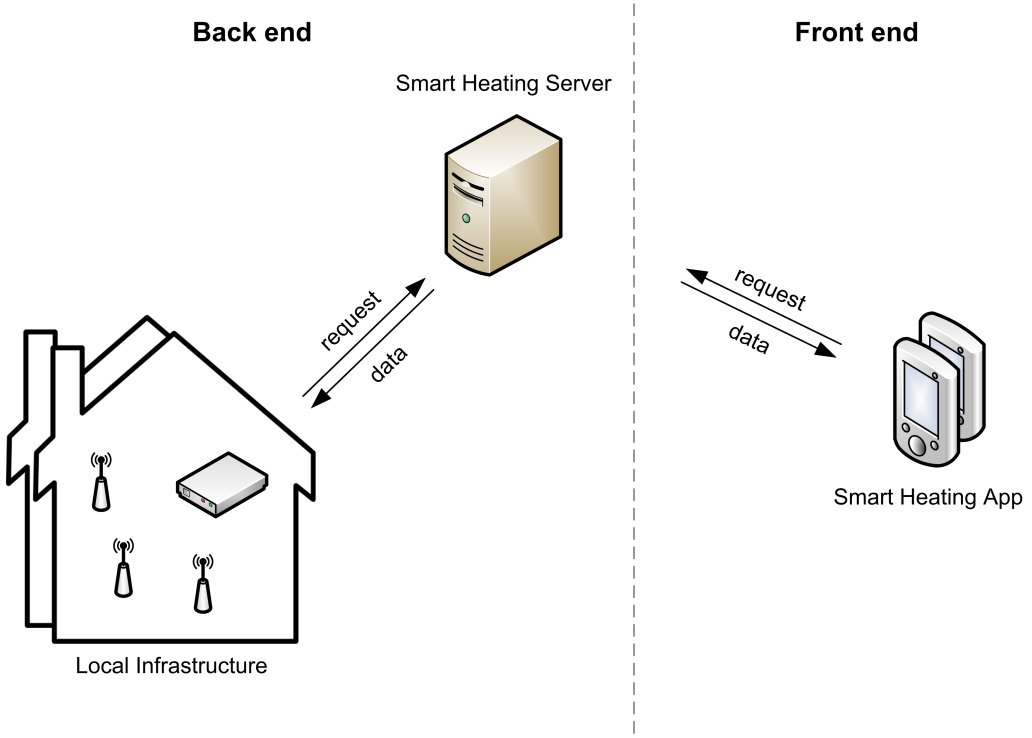

Figure 3.1: System overview showing the infrastructure part and the mobile application part. The local infrastructure is required for each residence and a residence can be controlled using one or multiple mobile applications. The server is required only once and stores the information about all residences.

The infrastructure part consists of the local residential deployment and the server. Within the residence a small computer system is installed which serves as a communication gateway to the distributed low-power thermostats. On the remote

side is the server acting as the central entity to collect and store the accumulated sensor data and organizational data. Both are explained in Chapter 4.

The mobile application part consists of a control application used to communicate between the user and the local infrastructure. It should provide an easy way for the user to control the smart heating system and guide him in installing it in his home.

3.1 Architecture

The smart heating system is implemented using a server-client model. The smart heating server hosts a web service acting as an API to provide access to the shared resources. The local communication gateway uses this API to query the residence configuration and persist the collected measurements. Complementary, there is the mobile application also acting as a client. It provides and updates the residence configuration and retrieves sensor data.

3.2 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.

The relationship between the entities could also be represented in a diagram. Think of visualising the use cases. These terms should match the ones in the use-case section.
SAMUEL: Falls du hier was passendes in den use cases hast ;)

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 4.2

User Each residence can contain multiple users. A user is associated to exactly one residence and is identified by his smart phone IMEI. 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.

4 Infrastructure

The back end is the system subset related to sensor reading, thermostat control and data persistence. The terms infrastructure and backend are used synonymously in this section. The infrastructure uses a server-client design pattern and is divided into the *server* side and the *local* side. On the one hand, there is the server as an independent unit providing a public API to its clients allowing them to communicate and share their data. On the other hand, there is the local deployment consisting of the installed wireless thermostats and the local communication gateway. Both sides will be explained in the following sections. See Figure 4.1 for an overview.

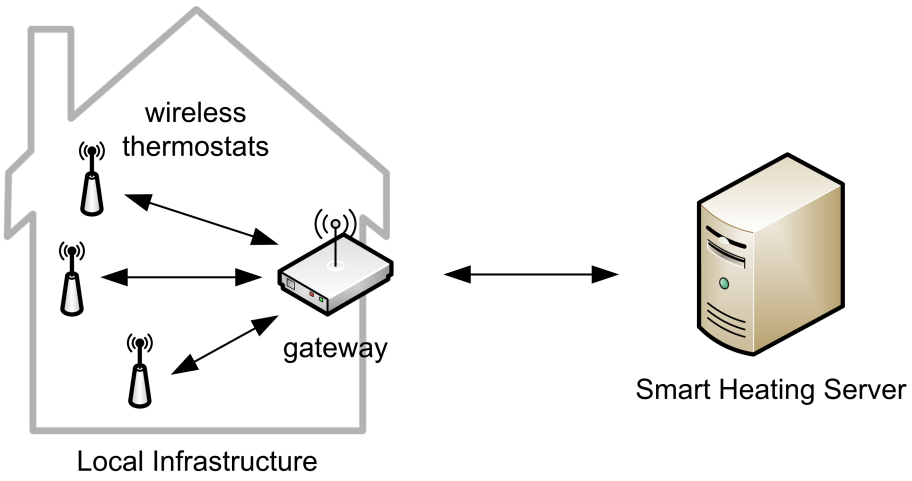

Figure 4.1: Overview of the backend consisting of the local deployment and the server infrastructure.

4.1 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.

4.1.1 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.

4.1.2 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 4.2 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.

4.1.3 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 based on 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

[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.

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.

A Uniform Resource Locator (URL) is a subtype of a URI that identifies resources via their location [1]. This API uses URLs as they are most common in the area of web services.

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[5]. Resources are referenced by including their representation.
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 4.1 for an example of a resource representation.

```
{
  "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 4.1: 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.

[5] Uniform Resource Locator (URL) is a subtype of URI. The details will not be explained here but the interested reader is referred to `https://en.wikipedia.org/wiki/Uniform_Resource_Locator`

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 overhead. See Figure 4.2 for an example.

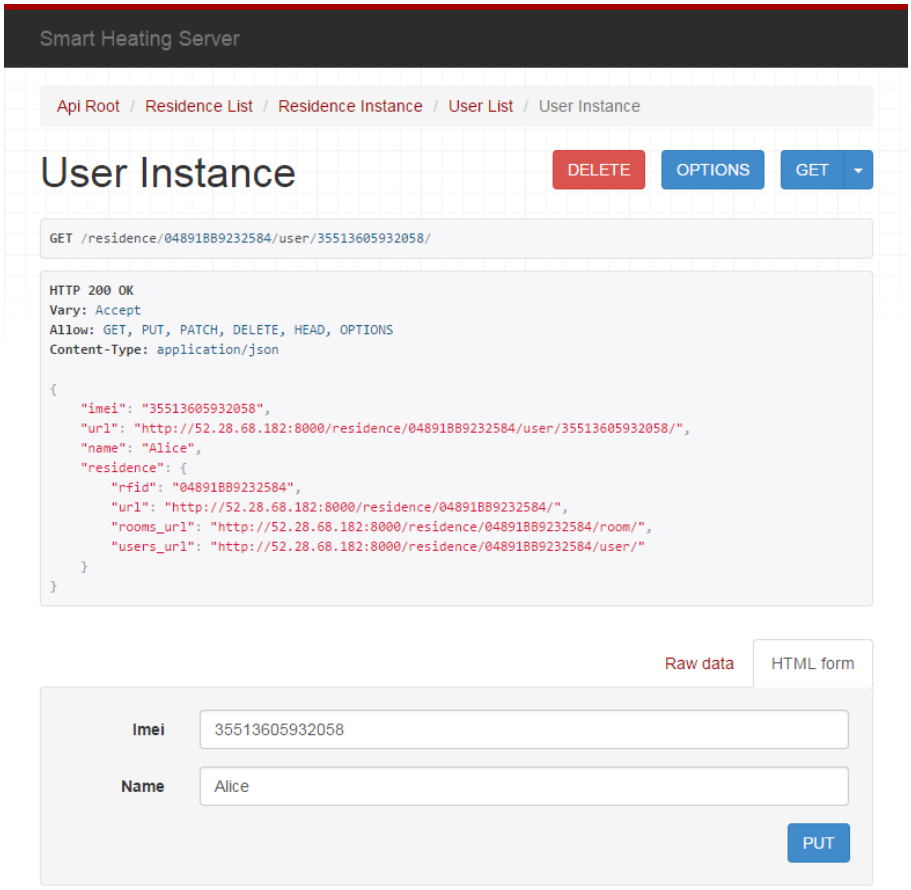

Figure 4.2: Screenshot of the Browsable Web API showing an instance of a user resource. The interface allows the developer to edit or delete the user. Navigating the residence or the associated room or user collections can be done by clicking on the URLs of the respective resources.

4.1.4 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[6].

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 4.1.5. The repository is designed to contain all code and dependency information necessary to automatically install and run the application on a server.

The following sections describe the application components in detail.

4.1.4.1 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 4.1.4.2

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 identifier of the radio-frequency identification (RFID) tag attached on the device. The mapping between RFID tag and MAC address is stored in the database, thereby making it easy for users to configure their system. See Figure 4.3 for a graphical representation of the used models. The most important models will be described in the following paragraphs.

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

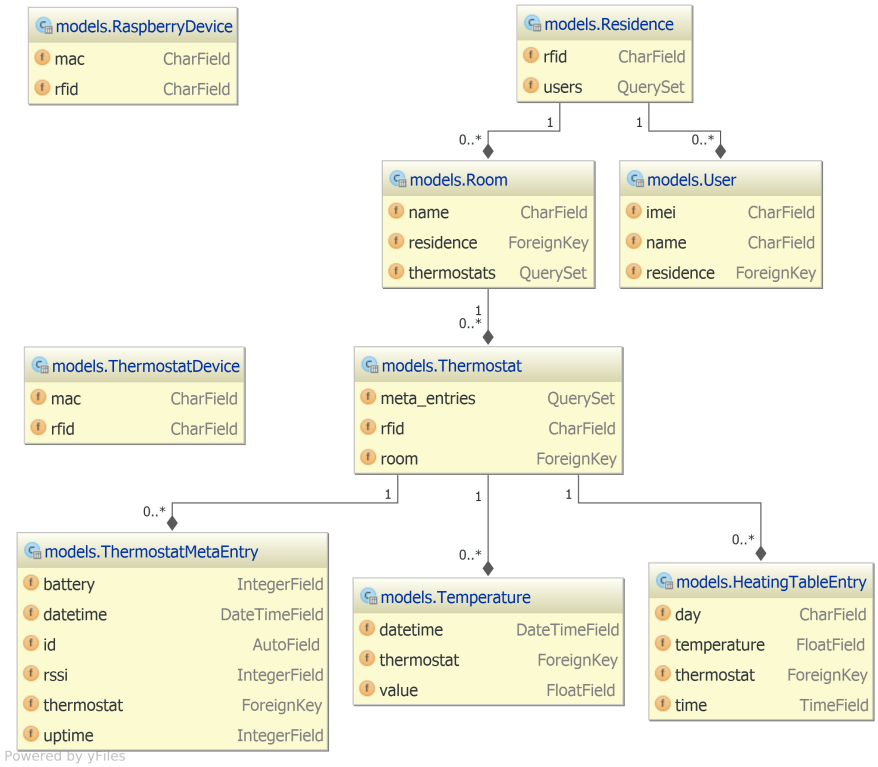

Figure 4.3: UML class diagram of the used models.

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[7] and can be registered to at most one residence.

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.

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

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.

4.1.4.2 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 4.2 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 4.2: The `RoomSerializer` class as an example of the usage of the `HierarchicalHyperlinkedIdentityField` and nested serializers.

4.1.4.3 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[8]. 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 4.3.

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

Listing 4.3: 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.

[8] 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`

4.1.4.4 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.

4.1.5 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 4.4 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.

Furthermore the usage of an Continuous Integration service like Travis CI[9] ensures that individual development branches are consistently tested and don't break the main development line upon branch integration. Additionally this convenient service ensures the periodic execution of all tests and logging of the test results.

4.2 Local Deployment

The local deployment 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

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

```
$ 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 4.4: Excerpt of the test output documenting the application behavior

master thesis as a basis to build upon [3]. 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 4.4 for an example of a local deployment.

4.2.1 Existing Infrastructure

This project builds upon work previously done by Nico Eigenmann as his master thesis [3]. Part of his work consisted of extending the hardware and software of the digital thermostat HR-20[10], as depicted in Figure 4.5, to offer wireless access via 6LoWPAN[11]. 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.

[10] `http://www.homexpertbyhoneywell.com/en-DE/Products/rondostat/Pages/HR-20.aspx`
[11] 6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Network

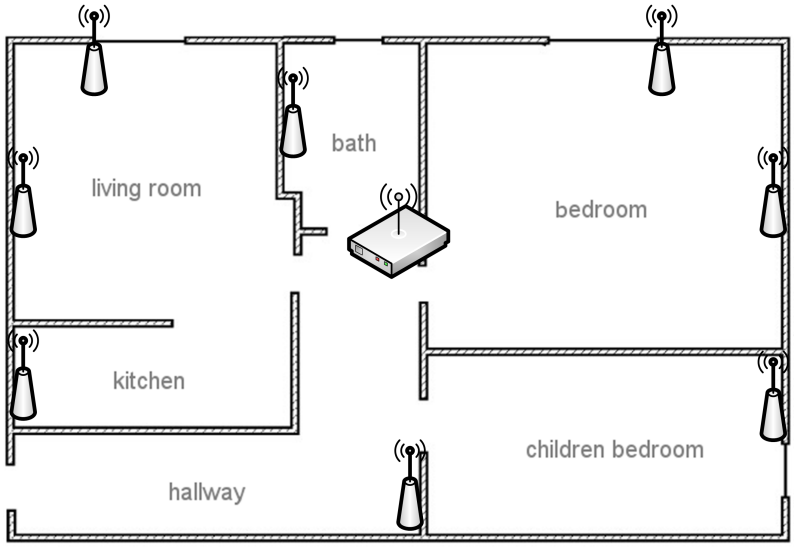

Figure 4.4: Example of a residence layout depicting a possible deployment. The local communication gateway is installed in the hallway, connected to the internet and has wireless connections to the deployed thermostats represented as antennas. Source of the original image: `http://www.haus-topplicht.de/wp-content/uploads/2013/12/planwohnung2.jpg`

Figure 4.5: Honeywell Rondostat HR-20 programmable thermostat. Source: `https://piontecsmumble.files.wordpress.com/2013/02/hr20.jpg`

4.2.2 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.

4.2.3 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) [4] 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. It is used to communicate with the API provided by the smart heating server.

4.2.4 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[12] as our local communication gateway which is responsible for two major tasks.

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.

4.2.4.1 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
```

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

The address with the prefix `fdfd:` is the relevant one accessible from the Raspberry Pi. 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.

Note: The discovered devices are not necessarily thermostats registered to the this local communication gateway. A discovered device can also be registered to another gateway or not be registered to any gateway, or it could also be any other unrelated device. The linked thermostats are therefore retrieved from the smart heating server to avoid unnecessary querying of unrelated devices.

4.2.4.2 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 4.2.4. The first script 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 4.2.4.3 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.

4.2.4.3 Local Storage

We combine SQLite and dbm for the local storage. *SQLite* is a lightweight relational database engine often used on embedded systems. It implements most of the common SQL-92 standard but omits some features like access permissions[13]. *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.

4.2.4.4 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

[13] https://sqlite.org/lang.html

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[14].

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

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

5 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 realised 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 5.1 shows some of the user interfaces for controlling smart heating systems.

In Section 5.1 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 5.2.2 through 5.2.5.

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 7, where possible future work on this project is listed.

Figure 5.1: Some examples of mobile applications for controlling smart heatings systems. Source: `https://cdn.recombu.com/media/digital/news/legacy/M13058/1397569835_w670_h576.png`

5.1 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.

5.2 Implementation

5.2.1 Application Flow

As seen in Figure 5.2 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.

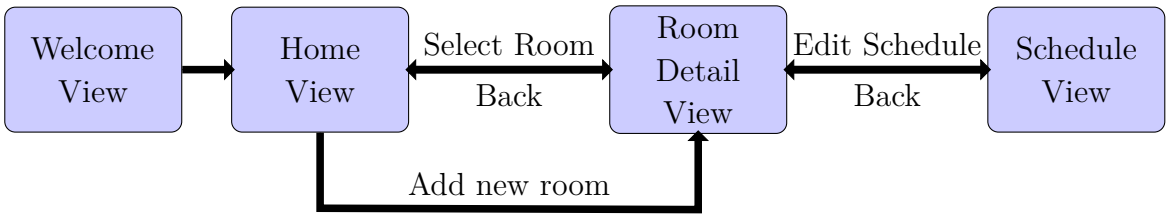

Figure 5.2: The application flow of the mobile app.

5.2.2 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 5.3. 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.

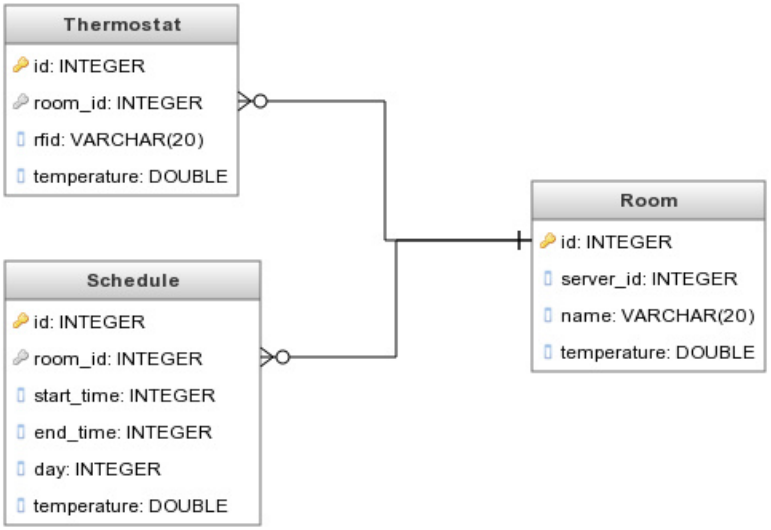

Figure 5.3: The database model used in the mobile application.

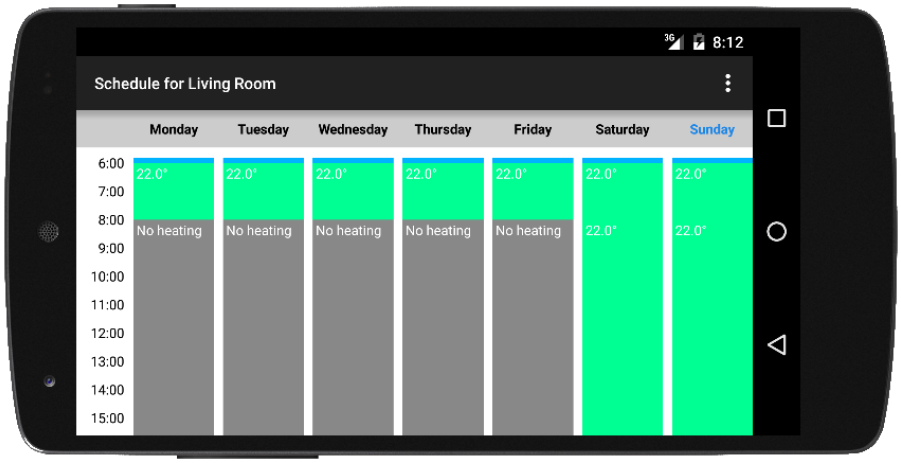

Figure 5.4: A sample default heating schedule after the initial setup of the system.

- 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 5.2.3).

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 5.2.5.

5.2.3 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 5.2.4. 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 5.5 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 5.2.4.

5.2.4 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 5.2.5.

picture detail view

5.2.5 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 5.6 for an example.

Figure 5.5: A sample home view with some random rooms.

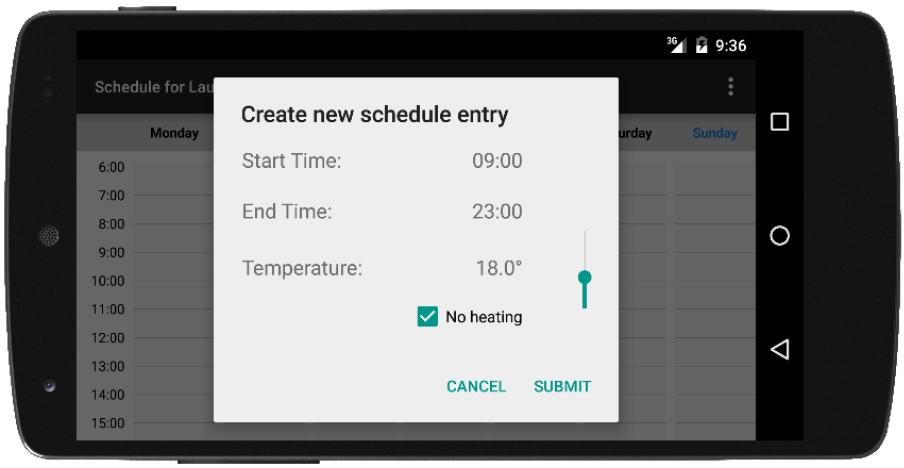

Figure 5.6: The popup for adding a new heating schedule entry.

5.3 Evaluation

5.3.1 Use cases coverage

Our mobile control application covers most of the use cases introduced in Section 5.1 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.

5.3.2 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...

6 Evaluation

In this section we will evaluate our set design goals and discuss our implementation in detail. Further we will analyze our design decisions. Where applicable the functional and non-functional requirements will be validated.

6.1 Infrastructure

This section follows the same structure as used in the previous Chapter 4. The infrastructure consists of the server part and the local part. Both are evaluated in the following sections. In the end the deployed infrastructure is evaluated in a field test.

6.1.1 Server Infrastructure

6.1.1.1 Design Goals

Section 4.1.1 lists the design goals defined for this project part. Each design goal will be evaluated in a separate paragraph below.

Modularity and Extensibility Django emphasizes reusability of components. This solid background allows us to clearly separate different concerns to achieve modularity. Further the distinct components simplify application extensions by easily adding new resources, representations, views or URLs.

Usability In order for a developer to be able to familiarize himself with a new API it is important to provide clear documentation and a platform to use the API. The browsable API as described in Section 4.1.3 provides both. It allows a developer to interactively explore the resource structure by navigating via the

displayed URLs. The browsable API also provides easy interaction possibilities to create, read, update and delete resources. Further the API describes the supported access methods when requesting a particular resource. One of these HTTP methods named *OPTIONS* shows the accessible fields, their types and other useful meta information for interacting with the API.

Testability The modular program structure and the chosen Django framework support the creation and automatized execution of software tests. These automatized tests are continuously executed on a hosted service and show that the tests run regularly and successfully.

6.1.1.2 Design Decisions

During the system implementation a few design decisions were chosen and will be evaluated in the following paragraphs.

Nested URL schema The model hierarchy is represented using a nested URL schema. This schema suits the model structure in tree form and induces good readable URLs. Django is designed for flat URLs but is still flexible enough to support the design decision of implementing a nested URL schema for resources access.

Resource identifier as the first field Within the resource representation the resource identifier is always the first field. This allows to easily recognize the identifier within resource representations and especially within nested resource representation.

Resource referencing Resources are referenced by including their representation. This design decision improves the usability of the API, as related resources are shown in their full representation and also allows to reduce the number of required requests. In contrast collections are referenced via their URL. This is necessary to limit the representation size and avoid infinite recursions. For example in a parent-child relationship the child representation contains the representation of its parent. The parent representation itself contains the collection representation of its children. Therefore the collection representation cannot contain the full representations of its resources.

Identification of URL fields Fields representing a URL are identified by the name `url` or the suffix `_url`. This design decision facilitates the recognition of hyperlinks which also allows an automatized navigation through the API.

6.1.2 Local Deployment

The following section will evaluate each of the design goals defined in Section 4.2.2. Please note that the reliability is evaluated separately in a field test described in Section 6.1.3.

The local infrastructure section is about the implementation part done on the residential communication gateway. Therefore the following paragraphs focus on the evaluation of software running on the Raspberry computer.

6.1.2.1 Performance

The system performance depends on two aspects. First, the software part. The programming language *Python* was chosen at it is suitable for developing on embedded hardware with possibly long-running input/output operations. Another important component is the system design of independent scripts triggered by a scheduler. Second, the embedded hardware part. The chosen hardware needs to be powerful enough to run the developed software implementation.

For evaluation of the ratio between required software performance and provided hardware performance the CPU load is monitored. We use the *sar* command from the *apt-get* package *sysstat* to analyze the average CPU utilization. Over a period of seven days of regular system runtime the average CPU load was 0.28 percent. This indicates that the hardware is powerful enough to run the designed software implementation.

6.1.2.2 Robustness

We defined robustness to be the ability of the system to behave reasonably in the presence of failures, especially connection problems. The local communication gateway is designed as two independent tasks. The first task is the retrieval and storage of the temperature data and the application of the defined schedule. The second task is the synchronization of the local data storage with the remote web server. This design splits the responsibility of the local communication system into

two simpler separated tasks with a clearly defined interface, the local data storage. In the presence of a failure only one of the two tasks is affected whereas the other task is still able fulfill its purpose.

For example in the event of a temporary internet connection disruption the system will keep operating the last downloaded heating schedule and store the temperature history. After the disruption the local data storage is synchronized and the system continues to operate on the most actual settings.

Another failure scenario would be the temporary disconnection of a wireless thermostat. In such a case the non-affected thermostats are still functional and the heating schedule of all thermostats is kept in sync with the web server. As soon as the system reestablishes the connection to the affected thermostat the system continues to work as planned. In the meantime neither the properly operating thermostats nor the synchronization of all thermostats' heating schedules is impaired.

6.1.2.3 Interoperability

We define interoperability as the system's ability to cooperate with other distributed systems. The whole infrastructure is designed to communicate according to recognized open standards and architectural styles such as *Constrained Application Protocol (CoAP)*, *Hypertext Transfer Protocol (HTTP)*, *Internet Protocol (IP)*, *JavaScript Object Notation (JSON)* and *Representational State Transfer (REST)*. This way the developed infrastructure is able to cooperate with other distributed system via the defined interfaces.

6.1.3 Field test

In order to check the reliability of the infrastructure in a real world scenario the system has been deployed in a residential home. Previous work in this area compared the defined room temperatures with the actual room temperatures to conclude on the proper work of a heating system [3]. Due to the summer season and the according temperatures this evaluation is not possible for this project. Instead we focus on the communication between the wireless thermostats, the local communication gateway and the remote server to evaluate the reliability. The main data sources for our analysis are the data storages on the local communication gateway, the database on the remote web server and the generated log files on both parts of the infrastructure. We present our analysis of two different time spans.

6.1.3.1 First evaluation

In the following we analyze the data from the time span of August 2015.

Local data storage analysis We start with the analysis of the local data storage. It is used to cache the temperature readings and other meta data read from the thermostats before it is send to the remote server. The local communication gateway queries the thermostats every 15 minutes, i.e. 96 times a day. The maximal number of temperature measurements and meta entries would therefore be 2976 for the whole month of August. The local data storage contains 2969 of these recorded temperature measurements in this date range, resulting in a coverage of 99.76 percent. The coverage of the meta data is even slightly higher. Please see Table 6.1 for the full coverage data.

Local Database entries	Maximal count	Actual count	Coverage
Temperature	2976	2969	99.76 %
Meta data	2976	2970	99.80 %

Table 6.1: Retrieved temperature measurements and meta entries in a real world deployment in August 2015.

Local log file analysis Each log entry has a severity level indicating the impact of a logged event on system stability and functionality. We evaluated these severity levels and their according log messages to analyze the system behavior as well as to identify potential problems.

There are almost a hundred thousand log entries which reveal a few interesting facts. The rate of successful CoAP requests varies highly depending on the queried resource. See the Table 6.2 for the full data. The requests to the operation mode and target temperature resources failed much more often than queries to other resources. Especially the PUT requests fail in more than 72 percent. Therefore the correct functioning could not be guaranteed in this time range.

We discovered the following potential issues:

- The applied target mode “manual target” often got overridden by “radio target”.

CoAP request	Total	Success	Timeouts	Success rate
GET /debug/heartbeat	2975	2970	5	99.8 %
GET /sensors/temperature	2975	2969	6	99.8 %
GET /set/mode	2975	2970	5	99.8 %
PUT /set/mode	803	181	622	22.5 %
GET /set/target	2975	1216	1758	40.9 %
PUT /set/target	2698	741	1957	27.5 %

Table 6.2: Analyzed CoAP requests in August 2015.

- There are many CoAP requests sent closely following each other. This may overwhelm the low power devices.
- There is an invalid temperature value in the heating schedule which is rejected by the thermostat. 1244 out of 1957 time outs are related to this invalid value.

We address these potential issues in the next section and describe appropriate improvements.

Another interesting fact is the handling of temporary losses of internet connection. Due to a internet modem failure and the necessary replacement actions, the residence local network was offline for about 48 hours. During this time the local gateway kept communicating with the wireless thermostat to operate the schedule and log the measured temperatures. After reestablishing the internet access all temperature measurements that have not yet been uploaded were pushed to the web server. It took about 56 seconds to upload the missing 192 measurements. During and after the whole external incident the system reacted and worked as planned.

Server analysis The server logs all received HTTP requests into a log file. Each log entry contains the requested URL, the HTTP method and the HTTP response code. In the chosen time span there were 2969 successful POST requests each adding a single temperature measurement. This matches the number of temperature entries in the storage on the local communication gateway and in the server database and shows that all temperature measurements were successfully transmitted and persisted on the server. In contrast the server log file shows that there is a known problem with the storage of meta data causing the related HTTP requests to be rejected. The local communication gateway makes five attempts until it marks an entry as failed. However no data is lost since marked entries are not deleted from the gateway database.

6.1.3.2 Second evaluation

After the long-term field test another test was set up to evaluate the effects of further improvements. In this second evaluation we focus our analysis on the CoAP communication between the gateway and the thermostat. We discovered a high failure rate for both PUT requests in the previous evaluation and tried to address this issue with several corrections.

We suspect the low power devices to may be overwhelmed by too many closely following requests. Therefore we drastically limit the number of request to one within three seconds. Additionally we repeat certain unanswered requests up to three times until a response is received. Furthermore the heating schedule is cleaned from the invalid value and the updated server does no longer accept temperatures that are not supported by the deployed thermostat.

The revised implementation is deployed to the local communication gateway and monitored. Due to time constraints we could not collect the same amount of data as in the first evaluation. The following analyzed data is based on a time span of 13.5 days. See Table 6.3 for the complete data.

CoAP request	Total	Unique†	Success	Timeouts	Success rate
GET /debug/heartbeat	1300	1300	1277	23	98.2 %
GET /sensors/temperature	1300	1300	1280	0	98.5 %
GET /set/mode	1300	1300	1286	14	98.9 %
PUT /set/mode	1	1	0	0	-
GET /set/target	1779	1300	1272	507	97.8 %
PUT /set/target *	49	49	6	40	12.2 %

† Number of requests not considering repetitions.

* Also see the paragraph *Hardware issues* for details.

Table 6.3: Analyzed CoAP requests of the second evaluation.

The curbed request rate and the repetition of timed out `GET /set/target` requests enhanced their success rate radically. The success rate increased from 40.9 percent to 97.8 percent. Interestingly the repetition of failed `PUT /set/target` requests did not improve its success rate. This adjustment was separately tested but then discarded. However the total number of PUT requests fell intensively due to the improved results of GET requests and the way of implementation, as PUT requests are only submitted if the desired values is not already set. The data shows the resource `/set/mode` only required a single write. Further analysis suggests that the

mode is also set when writing the target and does not need to be set separately. The remaining GET requests seem not to be affected by the taken improvements.

Hardware issues A closer look at the data indicates that the timed out `PUT /set/target` requests did still make a impact. Although when the gateway did not receive a response, the requested target temperature is returned by the subsequent GET query. Further inspection indicate that there is a general problem with the deployed Honeywell thermostat probably caused by weak hardware communication interfaces (UART) or the upgraded open source firmware OpenHR20.

6.2 Mobile App

SAMUEL: Mobile App evaluation

7 Future Work

In the scope of this lab project we developed a reliable heating control infrastructure and the corresponding innovative mobile application. We focused these implementations to build a solid basis for future projects to build upon. In the following we outline a few interesting ideas that we did not have the time to pursue but that could be implemented as a future work.

The developed system could be extended to track and predict user locations. This would allow to dynamically adjust the heating schedule to react to spontaneous user behavior deviations and could further decrease the energy consumption and increase the users comfort level in certain cases.

As a part of this project we implemented a mobile application designed for Android smart phones. Another possibility for future work would be an additional user interface that could show more complex data analysis suitable for devices with bigger screens.

SAMUEL: Future work mobile application

8 Conclusion

The goal of this project was to develop a reliable heating control system for residential users.

This project consists of two major parts. On the one hand there is the back end consisting of the local infrastructure monitoring and controlling temperatures and persisting this data on an accessible web server. On the other hand there is the front end mobile application offering a user friendly interface to configure and interact with the system.

During the development of this project a few issues occurred. One of the main obstacles at the beginning of the project was the lack of detail in the thermostat documentation. This resulted in a lot of time spent with debugging of the hardware and software components related to the chosen thermostats and their firmware. A further complication was the limited hardware performance which resulted in inconsistent request handling. For example, for certain requests the thermostat does execute but not acknowledge the request. We introduced a system that takes care of these anomalies. Our evaluation concludes that the adopted measures lead to a reliable heating control system.

TODO Samuel

Our proposed system of interacting but independent distributed systems provides a solid base for further developments in the area of residential smart heating systems.

Bibliography

[1] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), Jan. 2005. http://tools.ietf.org/html/rfc3986.

[2] Bundesamt für Energie BFE. Schweizerische Gesamtenergiestatistik 2013. Publikationsdatenbank, 2014. http://www.bfe.admin.ch/php/modules/publikationen/stream.php?extlang=de&name=de_208577679.pdf&endung=Schweizerische%20Gesamtenergiestatistik%202013 [Online; accessed September 30, 2015].

[3] N. Eigenmann. Opportunistic sensing for domestic smart energy systems. Master thesis, ETH Zurich, 2012.

[4] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014. http://tools.ietf.org/html/rfc7252.

[5] TEP Energy GmbH, Prognos AG, Basel, Infras AG, Bern. Analyse des schweizerischen Energieverbrauchs 2000 - 2013 nach Verwendungszwecken. Publikationsdatenbank, 2014. http://www.bfe.admin.ch/php/modules/publikationen/stream.php?extlang=de&name=de_523528791.pdf&endung=Analyse%20des%20schweizerischen%20Energieverbrauchs%202000%20-%202013%20nach%20Verwendungszwecken [Online; accessed September 30, 2015].

A First Appendix

A.1 GitHub repositories

All code developed in this project is published on GitHub and can be found at the following links.

- Code and documentation related to the local communication gateway running on a Raspberry Pi 2:
  `https://github.com/spiegelm/smart-heating-local`
- Code and documentation related to the server:
  `https://github.com/spiegelm/smart-heating-server`
- Code and documentation related to the mobile application:
  `https://github.com/Octoshape/smart-heating-app`

A.2 Setup instructions for the local communication gateway

This appendix explains the setup routine of the local communication gateways implemented on a Raspberry Pi 2. See also the repository documentation:
`https://github.com/spiegelm/smart-heating-local`

A.2.1 Setup raspbian

Flash raspbian to a SD card and boot the Raspberry. Find the IP address using `nmap -sP [ip-adress]/[bitmask]`, e.g. `nmap -sP 192.168.0.0/24`.

Open a SSH client and connect to the determined IP. The default username and password are `pi` and `raspberry`. Type `sudo raspi-config` to expand the filesystem, change the password and set the local time zone.

Enable IPv6: Add `ipv6` on a line by itself at the end of /etc/modules.

A.2.2 Install dependencies

Install `at`. Needed to remain tunslip6 started because UDEV rules kill the spawning process.

```
sudo apt-get install at
```

A.2.2.1 Install Python 3.4.1 and aiocoap

Credits to Marc Hüppin for the initial version.

openssl and libssl-dev are required for SSL support in python and is also required by pip.

```
sudo apt-get install sqlite3 libsqlite3-dev openssl libssl-dev
```

install the sqlite3 packages

```
mkdir ~/src
cd ~/src
wget https://www.python.org/ftp/python/3.4.1/Python-3.4.1.tgz
```

unpack cd into dir

```
./configure
make
sudo make install
```

get the aiocoap library from github: https://github.com/chrysn/aiocoap

get setuptools from: https://pypi.python.org/pypi/setuptools

install aiocoap using

```
python3.4 setup.py install
```

A.2.3 Setup smart-heating-local

A.2.3.1 Setup the border router connection

Clone this repository into your home folder:

`git clone https://github.com/spiegelm/smart-heating-local.git`.

Create symbolic links

- udev rules: `sudo ln -s /home/pi/smart-heating-local/rules.d/90-local.rules /etc/udev/rules.d/`
- tunslip executable:
  `sudo ln -s /home/pi/smart-heating-local/bin/tunslip6 /bin/`

Add this line to `/etc/rc.local` to make sure the udev rule is also executed on startup

```
udevadm trigger --verbose --action=add --subsystem-match=usb --attr
   -match=idVendor=0403 --attr-match=idProduct=6001
```

Reboot: `sudo reboot`

Attach the sky tmote usb dongle to the raspberry. The tun0 interface should be shown by `ifconfig`. In case of problems run `sudo ~/smart-heating-local/bin/start_tunslip.sh` manually. Determine the ipv6 address of the web service: `less /var/log/tunslip6`:

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

Retrieve the registered routes on the border router:

`wget http://[fdfd::212:7400:115e:a9e5]`:

```
<html><head><title>ContikiRPL</title></head><body>
Neighbors<pre>fe80::221:2eff:ff00:22d3
</pre>Routes<pre>fdfd::221:2eff:ff00:22d3/128 (via fe80::221:2eff:
   ff00:22d3) 16711422s
</pre></body></html>
```

Test route to the thermostat by requesting the current temperature via coap-client (libcoap):

```
~/smart-heating-local/bin/coap-client -m get
coap://[fdfd::221:2eff:ff00:22d3]/sensors/temperature
```

Congratulations, your raspberry is connected to a thermostat!

A.2.3.2 Install the required python packages

Install pip: `https://pip.pypa.io/en/latest/installing.html`

Install the project requirements:

```
cd  /smart-heating-local/
pip install -r requirements.txt
```

A.2.3.3 Configure cron tasks

Setup crontab to run the log and upload scripts periodically:

```
crontab -e
```

Insert these lines at the end of the file:

```
1 */15 * * * * /usr/local/bin/python3.4 /home/pi/smart-heating-local/
    thermostat_sync.py
2 */5 * * * * /usr/local/bin/python3.4 /home/pi/smart-heating-local/
    server_sync.py
```

These commands ensure that the temperature is polled from the registered thermostats each 15 minutes and checked for uploading to the server each 5 minutes. The scripts log interesting events to `~/smart-heating-local/logs/smart-heating.log`.

A.3 Setup instructions for the server infrastructure

Setup a virtual or dedicated server based on Ubuntu Linux 14.04 LTS and enter the following the commands in a terminal:

```
# Install dependencies
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install git python-virtualenv python3-pip
# Install code into home directory
cd ~
git clone https://github.com/spiegelm/smart-heating-server
# Setup virtualenv
virtualenv -p python3 env
. ~/env/bin/activate
# Check for python version >=3.4.0
python --version
# Install requirements, setup database and start server
cd ~/smart-heating-server/
pip install -r requirements.txt
./manage.py migrate
# Run server on port 8000
./manage.py runserver 0.0.0.0:8000
# Test that server is accessible via browser
# Kill server: CTRL-C
# Start server in the background using nohup
./scripts/restart_server.sh
# Kill background server
./scripts/kill_server.sh
```

In case of software updates run the following commands to fetch the newest code form the repository and restart the server:

```
cd ~/smart-heating-server
git pull
./scripts/restart_server.sh
```

A.4 Setup instructions for the mobile application

SAMUEL: add setup instructions for mobile application