

Magnus Bakke, Liang Zhu

Authentication in the Internet of Things

Bachelor's project in Information Technology with specialization
in Network administration

Supervisor: Stein Meisingseth

May 2019

Magnus Bakke, Liang Zhu

Authentication in the Internet of Things

Bachelor's project in Information Technology with specialization in
Network administration
Supervisor: Stein Meisingseth
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

AUTHENTICATION IN THE INTERNET OF THINGS

Prestudy report

Part of a Bachelor's thesis

**Presented to the Institute of computer technology and informatics
of the Norwegian University of Science and Technology
by Magnus Bakke & Liang Zhu**

Submitted in partial fulfillment for Bachelor's degree of
Informatics with specialization in network administration
during the year 2016–2019

Introduction

The internet of things (IoT) has been expected for many years, yet the technology, infrastructure and standards that preferably should come with it are still in their infancy. The newest protocols used for wireless communication in networks are considered secure, and it is easy to get *things* – devices, often small and without much of a user interface, that serve one or a few specific purposes, such as measuring temperature or cleaning floors – connected to a home network. But such devices are often manufactured specifically for home networks, rendering them incompatible with enterprise networks that typically do not and should not use the “one shared password” solution that home networks usually do. If these devices are to be connected to enterprise networks in a secure manner, the manufacturers need to design their products with such capabilities, or we have to find a solution that does not rely on the device’s specifications.

After our soon to be three year long period of studying Informatics with specialization in network administration at the Norwegian University of Science and Technology, which focuses heavily on projects in teams and team exercises, most have discovered that we work better with some than we do with others. We – Magnus Bakke and Liang Zhu, the authors of this prestudy – think we are an optimal match, and decided long ago that we should work together when the time came to write our Bachelor’s thesis. Magnus is an experienced programmer, and Liang has a talent for understanding, setting up and configuring networks and network components. We saw an opportunity to combine our strengths in one topic – a topic we find most interesting: Authentication in the internet of things. This assignment was suggested by Uninett AS, the state-owned company responsible for Norway’s National Research and Education Network, and revolves around getting IoT devices connected to enterprise networks securely and, preferably, conveniently. Their website is found at <https://www.uninett.no/>.

This document outlines our prestudy and will summarize the issues we face, needs and requirements, and our research into existing solutions. We will define goals for the projects, functional and non-functional properties and requirements, milestones and key activities, stakeholders and external conditions that might affect the process and/or outcomes of the project, success criteria and risks. This information should be sufficient to analyze the project’s financial feasibility, to define guidelines and standards for our process and documentation, and to make a plan for how we will organize the project if we find that it is feasible.

Background

WPA (Wi-Fi Protected Access, as originally defined in the IEEE 802.11 standard and later amended in IEEE 802.11i-2004) is a family of protocols for wireless authentication and security. There are two protocols in this family that are relevant today: WPA2-Personal and WPA2-Enterprise.

The vast majority of modern home networks utilize WPA2-Personal, which involves the use of pre-shared keys (PSKs), or passwords. When a guest arrives at a friend’s house and

wants to connect to the wireless network, the guest is given the password, which is entered on their device, and the guest is authenticated and connected by virtue of knowing the secret key. Because of its ease of use and configuration, and its relatively good security in small networks where all users can be trusted, this is a good solution for most home networks, provided that the password is complex enough (to prevent dictionary or brute force attacks, which enable an attacker to crack simple passwords) and the administrators/owners of the network are careful in respect to whom the password is shared with.

As per WPA2 (WPA3 has been announced and is said to support end-to-end between each device and the access point, even on open networks), this sharing of a “master password” means that any user on the wireless network can easily decrypt and monitor the communications of other users on the same network.

While WPA2-Personal uses PSKs, *WPA2-Enterprise* uses certificates or username/password combinations for authentication, authorization and encryption of communications. These credentials are unique to each user, and enterprise networks seldom have a master password. Using WPA2-Personal in an enterprise network would pose a significant security risk, with the risk increasing with every new user with access to the PSK; there is no good way of knowing when the PSK has been shared with persons outside the company, and there is no good way to trace activity back to a specific user (the only necessary piece of identifying information is the MAC address, which is easily spoofed).

Problems and needs of the industry

Needless to say, any solution for getting IoT devices connected to an enterprise network must first and foremost be secure. Conveniently, the WPA2-Enterprise standard allows us (and is designed) to encrypt the communications of each connected user, making it impossible for one user to monitor the communications of another. Inconveniently, manufacturers typically do not design their IoT devices to support the WPA2-Enterprise standard. In short, we cannot in any immediately obvious way connect IoT devices to an enterprise network without using a single pre-shared key (PSK), or password. Using a single PSK for all connected devices means that any person who wants to get an IoT device connected to the network must possess this secret key (unless the network is open, which as of now is out of the question if the network is to be considered secure). The fact that multiple persons possess the secret key means that any one of them can monitor the communications of all devices on the network. This is only secure if you can trust every person who is given the secret key. This property is implied by the “Personal” suffix; it is intended for home networks, and IoT devices are typically marketed to consumers for personal use in order to make their homes “smarter.”

The scenario we are presented with by Uninett involves students and faculty members bringing their own IoT devices to, for example, a campus. The student will expect to be able to connect the device to the wireless network. The device allows the user to input a password, but eduroam requires either a “username” (identity) *and* a password, or a certificate, neither of which are supported.

Uninett needs a deployable system that presents a personal wireless network to the user without the communications of the user's device(s) on that network being exposed to anyone but the user. The user should experience a personal wireless network through which they can communicate over the internet. Finally, the activity undertaken by users on the network must be traceable, meaning that communication must be associated with a user's identity.

Existing solutions

Surprisingly, there are few solutions that tackle this specific problem, and those that do exist are largely proprietary.

Multiple pre-shared keys

There are several variations of a technology that allows us to generate multiple PSKs for one SSID. This technology has been given names such as *multi-PSK* or *mPSK* (Aruba), *Dynamic PSK* or *DPSK* (Ruckus), *Private PSK* or *PPSK* (Aerohive) and *User PSK* or *UPSK* (Riverbed). These proprietary technologies involve taking some action (such as clicking a button, calling an API endpoint or filling out a form) in order to generate a new PSK which is given to the user who wants to connect a new device. The generated PSK will be completely new and unique. The user can use the newly generated PSK in the same way as they would use a "master password," but the communication between the device and the access point will be encrypted using this specific PSK. Because no two users will possess the same PSK, they cannot monitor each other's communications.

This technology is used by SURF, an organization that offers ICT services for Dutch education and research, much like Uninett does in Norway. An entry written by Sven de Ridder on the organization's blog describes using Aerohive's PPSK technology for connecting IoT devices to a wireless network.

If we are to use these technologies to solve the problem of connecting IoT devices to enterprise networks securely, we must obtain the necessary software and hardware, possibly through the trials sometimes offered for evaluation purposes.

Hotspots/connection sharing

Most modern phones allow the user to share their data with nearby devices. This is popularly referred to as enabling a hotspot. Major operating systems for laptops and desktop computers also support this functionality. The user may customize the name of the hotspot and the password required for connection.

When a device connects to such a hotspot, the phone or computer forwards and receives communication on the connected device's behalf, which is encrypted between the hotspot-enabled device and the connected device.

While we do not know of any large scale solutions utilizing this technology, this may be considered an existing solution to IoT connectivity, as it allows a user on an enterprise

network to broadcast a private hotspot (with its own unique PSK) for their devices, which may not support WPA2-Enterprise.

The challenge is to make a streamlined solution, which will most likely rely on dual-band Wi-Fi adapters, some automation through scripting, and development of graphical user interfaces, all contained within a distributable box, such as a Raspberry Pi.

Definitions

<i>prestudy</i>	A study involving activities such as research, planning and charting/analyzation of risks, possibilities, necessities and stakeholders of a project
<i>prestudy report</i>	A report summarizing the prestudy
<i>PSK</i>	Pre-shared key; a password for a wireless network
<i>purpose</i>	What we, the candidates, hope to achieve with the project; <i>effect goals</i> ; not necessarily quantifiable
<i>process goals</i>	Goals, usually quantifiable, pertaining to how the project is executed
<i>performance goals/targets</i>	In this context: Specific, quantifiable/measurable goals pertaining to the properties of the final product, benefits achieved, costs and similar
<i>IoT</i>	<i>Internet of things</i> ; popular term for an internet that is – to a high degree – populated by autonomous devices often not controlled by a user, such as smart thermometers, refrigerators, vacuum cleaners and sensors
<i>eduroam</i>	World-wide roaming access service for research and education

Table of contents

Introduction	2
Background	2
Problems and needs of the industry	3
Existing solutions	4
Multiple pre-shared keys	4
Hotspots/connection sharing	4
Definitions	5
Table of contents	6
Purpose and goals	8
Purpose and selection of goals	8
Process goals	8
Performance goals	8
Scope and progression of the project	10
Functional properties and requirements	10
Non-functional properties and requirements	10
Project milestones and key activities	10
Key activities	10
Milestones	12
Phases	13
The research phase	13
The design phase	13
The execution phase	14
The finalization phase	14
Progress plan	14
Gantt chart	15
Introduction to possible solutions	16
Multiple pre-shared keys	16
Requirements	17
Benefits and drawbacks	17
Pre-configured hotspot devices	18
Requirements	19
Benefits and drawbacks	19
Stakeholders, external conditions and regulatory frameworks	20
Stakeholder analysis	20
External conditions and regulatory frameworks	22
Success criteria	23
Common success criteria	23

Success criteria for the hotspot solution	23
Success criteria for the multi-PSK solution	23
Plan for effective communication	24
Risk analysis	25
Risks	25
Severity and likelihood of events	25
Risk threshold	25
Definitions of levels	28
Estimation of severity and likelihood of risks	29
Graphical representation	30
Risk management	30
Analysis of financial feasibility	32
Non-quantifiable benefits	32
Financial feasibility of the hotspot solution	32
Conclusion	33
Financial feasibility of the multi-PSK solution	34
Conclusion	35
Guidelines and standards	36
Documentation requirements	37
Quality control requirements	38
Controls and methodology	38
Methodology	39
Standards and principles	39
Controls	39
Project organization	39
Conclusion	40
Amendments	41
Bibliography	42

Purpose and goals

Purpose and selection of goals

We will define a set of goals for this project. The selection of these goals will be based on our understanding of Uninett's needs, as well as our best judgment. The selected goals should reflect the project's *purpose* or *effect goals*, which we state thusly:

- To give the reader of our reports a high degree of insight into the problems we face in connecting IoT devices to enterprise networks securely
- To enable the reader to easily replicate our solutions
- To prepare the reader for the transition into a highly IoT-connected world in a satisfactory way
- To reduce the reader's costs and time spent on research on the topic

Furthermore, we will categorize our goals. We define the following two categories, in addition to the stated purpose or effect goals:

Process goals

Goals pertaining to the way we complete the project, including timeliness, attendance, and the quality and scope of documentation.

Performance goals

Goals pertaining to the properties of the solution and/or report to be developed; functionality, quality and deadlines. These goals are defined by Uninett.

Process goals

The following is a combination of process goals defined by NTNU, Uninett and the candidates:

- 500 (+/- 5%) hours spent per candidate
- Zero disputes/conflicts between the candidates
- No deviations from the progress plan exceeding three days
- Project completion at least three days before the deadline on May 20th
- Documentation of every major activity and decision

Performance goals

In addition to satisfying the requirements in regards to functional properties, we have set the following performance goals:

- Achieve internet connectivity for IoT devices in less than two minutes

- Achieve an average upload and download speeds greater than 80% of those normally achieved on Uninett's network
- Experience no more than 120% the amount of drops to less than 20% of the average download/upload speed compared to a non-IoT connection

Scope and progression of the project

Functional properties and requirements

As agreed upon by us and Uninett, the solution to be designed and prototyped shall have the following functional properties:

- Users may bring their own IoT devices to the premises and achieve internet connectivity
- The user responsible for abuse should be identifiable
- The communication between a device and the access point should be encrypted and visible only to the user who connected the device

Furthermore – either in addition to or as a consequence of the functional properties – the following requirements apply to the solution's functionality:

- The solution must allow the user to authenticate using their institutional credentials
- Each user must be given a unique PSK
- PSKs must be generated automatically on demand

Additionally, we hope to achieve the following:

- Roaming between access points
- A custom dashboard for requesting PSKs

Non-functional properties and requirements

The following requirements also apply, and pertain to aspects of the solution that do not directly relate to functionality and the way the solution behaves:

- The solution should be financially feasible
- The solution should not cause harmful interference
- The solution should not cause excessive administrative work
- The solution should not require significant changes to network architecture
- The solution needs not be ready for large-scale deployment, but it should be easy to prepare for such deployment

Project milestones and key activities

Key activities

There are 9 key activities in this project:

1. Research

Researching existing technologies and possibilities, charting needs and problems, and laying out the gathered information for purposes like planning and design is arguably the phase of

this project that will lead to the most severe consequences for later phases if done poorly. Proper research must be made before we can comfortably begin to design solutions.

2. Prestudy and writing of the prestudy report

The prestudy is an activity that involves making plans, assessing risks, analyzing stakeholders, and summarizing our research. The prestudy *report* is written as the prestudy takes place, and summarizes our plans, conclusions and findings in a report format. This document is the prestudy report.

3. Design and writing of the design report

Once the prestudy is complete, we will have a good idea of which solutions we will be going forward with. These solutions need to be carefully designed. A thorough and good design will be invaluable when the time comes to develop the solution and put it to the test. The design will include UML and UX charts and diagrams, as well as mockups of user interfaces, technical blueprints and similar. All documents and plans produced during this activity will be included, explained and argued in a design report.

4. Prototyping

Once the designs are complete, they can be implemented in a working prototype. This activity involves writing code, setting up any necessary network infrastructure, actualizing user interfaces and similar. The process will be described in the execution report.

5. Testing and revision

Bugs and potential for improvement will be identified, and the prototypes will be changed accordingly. The process will be described in the execution report.

6. Writing of the execution report

The execution report will be an extensive document describing the process of implementing the designs, developing and testing prototypes, issues encountered, solutions, evaluations and similar. The prestudy report, the design report and the execution report will be included in the final report.

7. Writing of the final report

The final report will include the three existing reports, as well as a preface and conclusion. This is an activity of shorter duration.

8. Planning for the presentation

A presentation marks the end of the project. This activity involves preparing demonstrations of functionality, planning topics, rehearsing and similar.

9. Finalization

Once the necessities of the project are in place, we may decide to improve the formatting of documents, make minor changes to user interfaces to improve aesthetics and user friendliness, optimize code and similar. This activity does not constitute a critical success criterion, and will only be carried out when and if all crucial documents and activities are done.

Milestones

Based on these key activities, we can define the project's milestones thusly:

1. Completion of research

When this milestone is reached, we intend to have conducted rudimentary tests of required technology to guarantee that our initial plans will indeed be possible to design and execute. We also hope to have eliminated as much uncertainty as possible, which will include ensuring that we will have access to any proprietary technology necessary.

2. Completion of the prestudy

Once the research is completed, the gathered information must be summarized and presented in a report format. The prestudy report will also provide an outlook for the remainder of the project period with plans and analyses. This activity will be undertaken partially in parallel with the initial research, and the prestudy report will become more populated as more information is gathered. The completion and approval of this document marks the end of this activity.

3. Completion of designs

When our solution is fully designed, we will have a clear picture of the road ahead. When this milestone is reached, we anticipate to have eliminated almost all uncertainty; any flaw in our plans should be made obvious by our diagrams and charts, and we should know exactly what components are missing and what code needs to be written.

4. Completion of the design report

The design report should present our designs so that Uninett knows what to expect of our final product. Readers of the report should also have a clear understanding of what the solution will look like to the user.

5. Completion of prototypes

Working prototypes will be proof of the viability of our planned solutions. Bugs, shortcomings and potential for improvement will become apparent, and Uninett will be able to try and comment on the solution before improvements are made.

6. Completion of revisions

Once suggested improvements are made, bugs are fixed and all integration tests have passed, the solution will be acceptable, the user experience should be pleasant, and the product should be easy to deploy.

7. Completion of the execution report

The finalization of the execution report marks the end of the execution phase. It should give the reader a clear understanding of the process of actualizing the designs, and should demonstrate that the solution works as intended. It should also teach the reader how to deploy the solution.

8. Completion of the final report

The final report is the last document to be written. It should give the reader a clear

understanding of every step of the project, from initial ideas and considerations, to the design, development and deployment of a working solution.

9. Presentation

The presentation of the final product is the last of the project. The presentation should give the audience a generalized idea of the processes that led to the final product, as well as an overview of the properties of the product itself and the problems it solves.

Phases

We have divided the project period into four primary phases. These phase are: 1) The research phase, 2) the design phase, 3) the execution phase, and 4) the finalization phase.

The research phase

The research phase includes the following activities:

- Research
- Prestudy
- Writing of the prestudy report

The goals of the research phase are to:

- Gain a deeper knowledge of the topic
- Gather information on existing solutions
- Chart necessities
- Develop ideas and find plausible solutions
- Gain an overview of dependencies
- Plan project phases
- Develop initial proof of concept
- Summarize findings, decisions and plans in a prestudy report

The design phase

The design phase includes the following activities:

- Design
- Writing of the design report

The goals of the design phase are to:

- Plan information flows
- Finalize proof of concept
- Plan software architecture
- Design graphical interface mockups
- Have a strong basis for execution/development

The execution phase

The execution phase involves the following activities:

- Development of prototypes
- Testing
- Writing of the execution report

The goals of the execution phase are to:

- Develop software in accordance with designed architecture
- Develop graphical interfaces
- Connect graphical interfaces with developed software
- Perform integration tests

The finalization phase

The finalization phase involves the following activities:

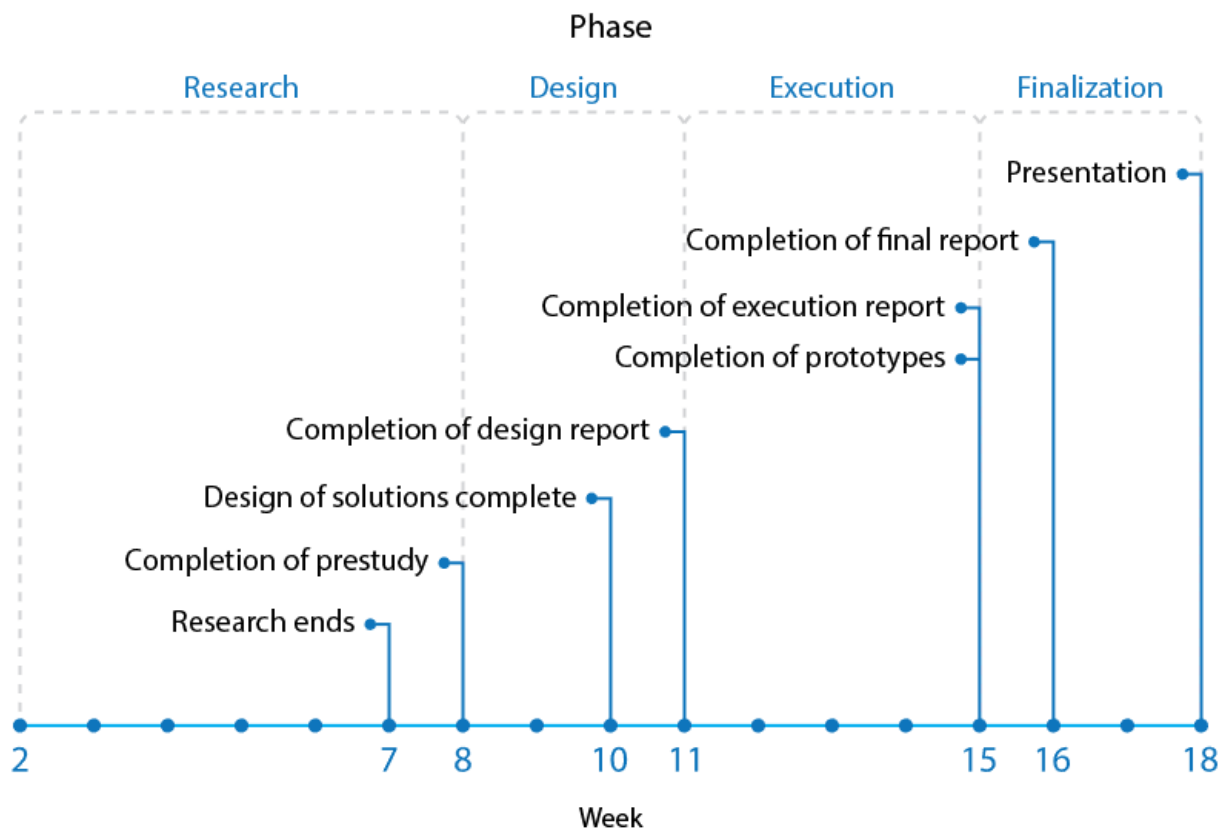
- Writing of the final report
- Planning for and holding the presentation

The goals of the finalization phase are to:

- Format documents according to the requirements for documentation as described in this document
- Have manageable and efficient code
- Have aesthetically pleasing user interfaces
- Simplify the deployment process
- Be prepared for the presentation
- Leave a good impression and efficiently communicate the importance and efficiency of the solution

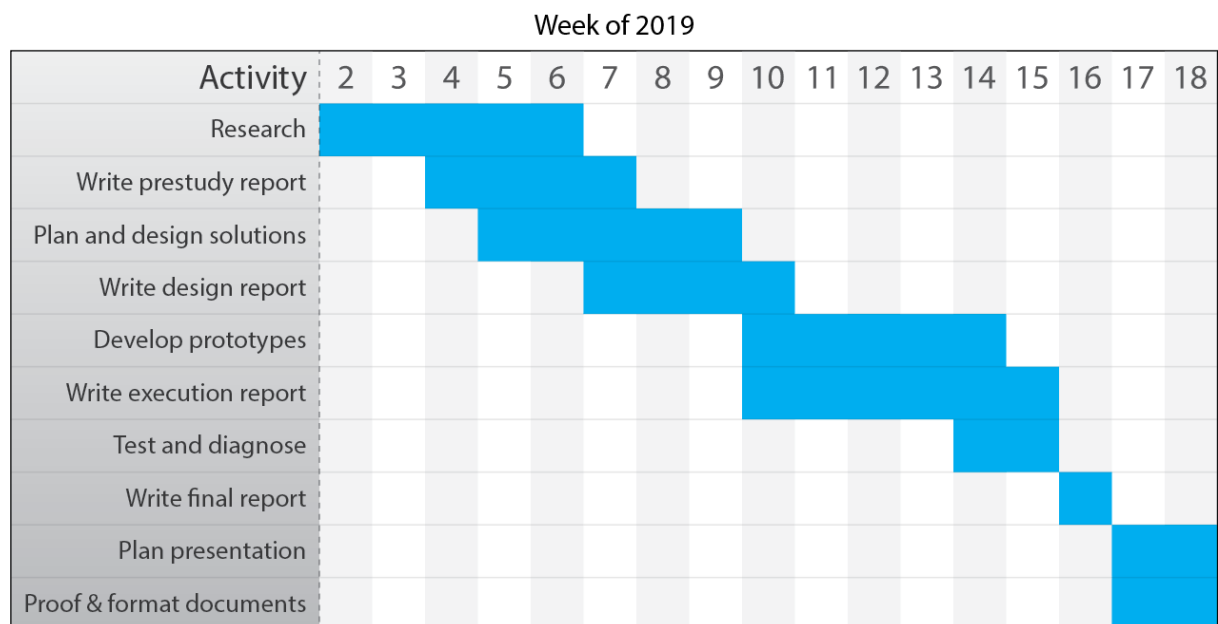
Progress plan

We have made this illustration to show when we intend to reach the major milestones, and which phase these milestones are part of. The milestone is meant to be reached at the transition from one week to another, meaning that a milestone on week 7 should be completed by midnight on the Sunday of week 6.



Gantt chart

Furthermore, we have made this Gantt chart to show when we intend to work on each key activity:



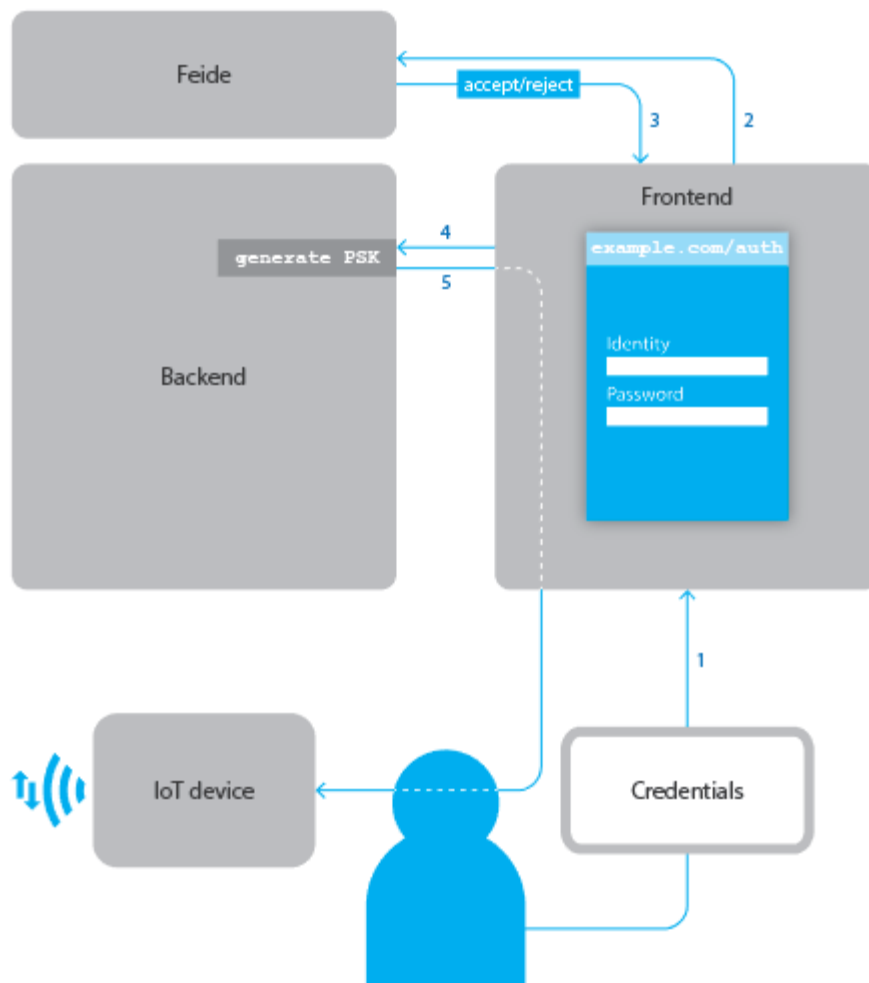
Revisions may be made.

Introduction to possible solutions

Our research has revealed two possible solutions. When we speak of “the multi-PSK solution” and “the hotspot solution,” these are

Multiple pre-shared keys

Our preferred solution involves end-to-end encryption using multiple pre-shared keys. The generation of said keys could be done as shown in the below illustration.



The user inputs their credentials into a frontend – a web page that asks the user for their Feide¹ account’s username and password. If the credentials are accepted by Feide, an API endpoint or method is called. This endpoint generates a new PSK, which is displayed on the web page to the requesting user. The user may then use this PSK for their device(s). Additional security measures, such as event logs, PSK expiration and limitations on the number of devices allowed on a single PSK may be implemented.

¹ "Feide | Sikker innlogging og datadeling i utdanning og forskning." <https://www.feide.no/>. Secure login and information sharing for education and research.

Requirements

This solution requires access to one of the mentioned providers' proprietary software, as well as a compatible access point. Optionally, we may use *hostapd*, a daemon for wireless access point and authentication servers that supports MAC-specific PSKs, to simulate such proprietary software.

In addition, the solution requires a backend – the service that allows you to generate a new PSK (preferably provided by the AP controller software), a frontend that allows the user to provide credentials, and Feide integration for validation of said credentials.

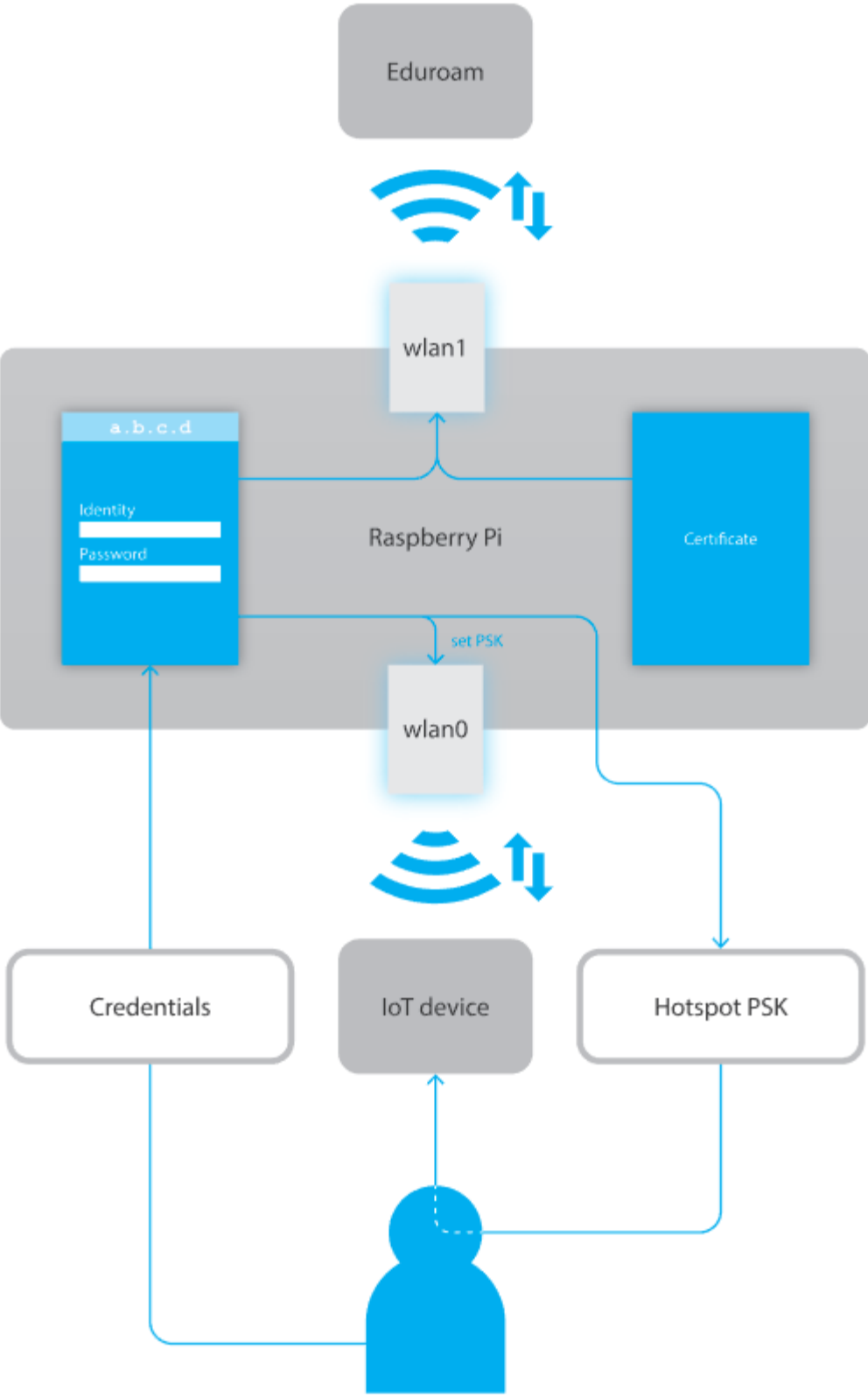
Benefits and drawbacks

This is a highly secure, practical and elegant approach, in our opinion. Depending on the specifics of the implementation, it may allow the administrator to enable group PSKs – passwords that work for more than one device. With group PSKs disabled or not configured, one password would work for one device only. This makes the system highly flexible.

The main drawback is the cost of implementation, as the solution requires an upgrade of existing access points (Uninett's current APs do not support multiple PSKs). In addition, a trial seems impossible to get from any provider of this technology: We have tried to contact all of them, but none have responded to our inquiries

Pre-configured hotspot devices

The below illustration shows one possible flow when using hotspot devices for getting IoT devices connected to eduroam.



Requirements

The development and testing of this solution requires at least one Raspberry Pi or Arduino with two radio antennas. The dual antenna can be achieved with a supported Wi-Fi adapter, such as the Asus USB-N10.

Benefits and drawbacks

This is a cheaper solution, and one with more predictable costs. The cost is at most the price of a Raspberry Pi plus the cost of a compatible Wi-Fi adapter per *user* (not per device) who wishes to bring IoT devices to the network. The solution would be relatively simple to develop and deploy.

Raspberry Pi machines are known for their instability, however, and continuous operation would almost certainly be problematic. In addition, a high number of “rogue” hotspots may be undesirable. There is also a possibility of theft of or damage to the devices. The fact that the software runs on the device, which is accessible to the user, makes it vulnerable to exploitation in the form of wiping the drive and using it for other purposes.

Stakeholders, external conditions and regulatory frameworks

Stakeholder analysis

This project has few critical stakeholders, and the stakeholders that do exist have clearly defined success criteria and demands. We have identified the following project stakeholders.

	Stakeholder	Success criteria	Contribution
Internal	Candidates	<ul style="list-style-type: none"> - The product meets the criteria - The final report is of high quality 	<ul style="list-style-type: none"> - Research and development
	Advisor (NTNU)	<ul style="list-style-type: none"> - Uninett's requirements are met - The final report is of high quality - 500 (+/- 5%) hours per candidate - The project is finished on time 	<ul style="list-style-type: none"> - Project (not product) criteria - Oversight
	Advisor (Uninett)	<ul style="list-style-type: none"> - Uninett's requirements are met - The product is finished on time 	<ul style="list-style-type: none"> - Expert advice
	Uninett AS	<ul style="list-style-type: none"> - Requirements are met - Cost efficiency/financial feasibility 	<ul style="list-style-type: none"> - Product requirements - Office and facilities - Professional contacts
External	Users	<ul style="list-style-type: none"> - Ability to get IoT devices online securely - Ease of use 	<ul style="list-style-type: none"> - Demand for product
	Administration	<ul style="list-style-type: none"> - Little extra administrative work - Clearly defined administrative tasks 	<ul style="list-style-type: none"> - Distribution of hardware and keeping track of users (administrative work)
	Regulatory bodies	<ul style="list-style-type: none"> - Compliance with regulations pertaining to security and privacy 	<ul style="list-style-type: none"> - Incentive to maintain security and privacy
	Legal entities	<ul style="list-style-type: none"> - Compliance with regulations pertaining to security and privacy 	<ul style="list-style-type: none"> - Legal responsibility

We define *internal stakeholders* as those who are directly involved with the project, and *external stakeholders* as those who are not.

The purpose of this stakeholder analysis is to identify and deal with the challenges (including resistance to the project) possibly faced with each stakeholder. The analysis should also enable us to better satisfy the stakeholders' expectations.

Note that, because this project only involves developing a prototype and not a full scale implementation, there is very little risk and resistance involved. A full-fledged implementation might deal with resistance from certain employees against increased responsibilities, for example, but that will not be relevant in our case. The scope of the project is to simply develop a reference for further development and implementation.

While the administration and users will not be affecting this project, we would like to develop a solution that all stakeholders will support should the solution eventually be implemented at scale. We will attempt to do this by minimizing or eliminating additional administrative work and conducting user tests.

It is crucial that the solution is in compliance with regulations. Regulations that affect this project dictate the minimum level of security and privacy that the solution should offer. We will ensure compliance by verifying the proper encryption of traffic, by storing only strictly necessary data about users' behavior (and storing this data only for as long as it is needed for the system to function), and securing the backend.

As for the high-power, high-interest stakeholders, we have planned to hold regular meetings with them in order to make sure their needs are satisfied. They will have the opportunity to influence the solution heavily. We are in close contact.

Advisor Stein Meisingseth from NTNU also takes part in these meetings, and we will make sure his success criteria are met.

External conditions and regulatory frameworks

There are two categories of regulatory frameworks to consider in this project: 1) Legislation, and 2) the policies of eduroam. The policies of eduroam are relevant only if the hotspot solution is implemented, as the hotspots will act as a bridge between the user's device and eduroam. If the multi-PSK solution is implemented, the user's devices are not connected to eduroam, but rather a local network.

Legislation

The Norwegian Personal Data Act, the directive on the processing of personal data and the directive on employers' access to email inboxes and other electronically stored material regulate the extent to which we are allowed to store user data, as well as for how long and what this data may be used for. No sensitive user data (such as social security numbers) will be collected. In fact, only a register of which PSK or hotspot has been leased to which user is necessary. Even this information will be deleted once it is no longer necessary.

The directives also regulate companies' opportunity to view the personal data and communications of employees and users. In principle, doing so must be strongly justified. Ensuring the proper functioning of systems is one such justification. (Data seen by network administrators in this context should not be shared with the employer.) It is necessary and justified to allow network administrators to view a list of sessions/leases, so that they may be terminated or diagnosed.

It is not justified to let network administrators view traffic. The solution developed should not feature tools that make it easier to monitor the communications of users.

Eduroam policy

eduroam's *conditions of use* is a short disclaimer of responsibility that does not make any statements about the use of hotspots or multi-PSK technology. eduroam's websites state that eduroam providers should not make use of web portals — websites used for authentication

— because eduroam uses certificates, which are in themselves the strongest and least vulnerable form of authentication. This is not part of their conditions of use, however; it merely a statement saying that eduroam “does not use web portals.” This consideration will be moot unless we proceed with the hotspot solution, as Feide uses a single sign-on page, so we need not and cannot require that the user enters their Feide credentials on our own frontend. In other words, if we require Feide authentication, which will be the case if we proceed with the “multi-PSK” solution, these will never pass through our system. If we require eduroam credentials, however, as will be the case if we proceed with the “hotspot” solution, these must be entered on a local page running on the disconnected hotspot device.

Success criteria

The “hotspot” solution and the “multi-PSK” solution have different success criteria. We have defined success criteria that apply to both, as well as success criteria that are unique to each solution.

Common success criteria

The following success criteria apply to both the hotspot solution and the multi-PSK solution:

- The project is completed before May 20
- Users may successfully connect devices that do not support certificates to a wireless network
- Abuse on the network should be traceable
- The communication between the access point and the device is encrypted with a key known only to the device’s owner
- The PSKs generated must be sufficiently random so that they cannot be predicted
- The solution can be replicated by reading our reports

Success criteria for the hotspot solution

The following success criteria apply to the hotspot solution only:

- Once configured, the hotspot needs only be reset in order to ready it for the next user
- The user needs only enter their Feide credentials

Success criteria for the multi-PSK solution

The following success criteria apply to the multi-PSK solution only:

- The user does not need to know the device’s MAC address for access to be granted
- It must be possible to support different multi-PSK technology providers by simply replacing a module/component of the solution
- It should be possible to generate PSKs through a customized frontend.

Plan for effective communication

To facilitate effective communication between stakeholders, we have devised the following communication plan. It describes who is responsible for communicating which message to which audience through which channel at what time/how frequently, and feedback channels.

Audience	Message(s)	Media/vehicle(s)	Frequency/timing	Provider/messenger	Feedback mechanism(s)
Uninett, NTNU	Status and plans	Face-to-face meetings	Biweekly (10:00 AM on Wednesdays)	Candidates	Comments during meeting; e-mail
Uninett, NTNU	Detailed progress/weekly reports	Document in shared folder in cloud storage	Weekly	Candidates	E-mail, biweekly meetings
Uninett, NTNU	Progress plans	Document in shared folder in cloud storage	Weekly	Candidates	E-mail, biweekly meetings
Uninett, NTNU, candidates	Meeting minutes	Document in shared folder in cloud storage	After every meeting	Candidates	E-mail, next biweekly meeting
Uninett, NTNU	New weekly report, progress plan and meeting minutes available	E-mail	Weekly	Candidates	Biweekly meeting
Candidates	Changes requested to solution or design	Bi-weekly meetings or e-mail	As needed	Uninett	E-mail, face-to-face discussion
Uninett	Requested changes implemented	Bi-weekly meetings or e-mail	Upon completion	Candidates	E-mail, face-to-face discussion
NTNU	Hours spent (with activity descriptors)	Work log (spreadsheet)	As requested	Candidates (separately)	N/A
Uninett, NTNU	Delays/unexpected issues	Weekly report in shared folder	When applicable	Candidates	E-mail, face-to-face meetings
Uninett	Critical issue(s)	E-mail, face-to-face meetings, crisis meetings*	When necessary	Candidates	Crisis meetings*, face-to-face meetings

Risk analysis

One goal of the prestudy is to minimize uncertainty. In order to do this, we must minimize risks, and before we can minimize risks, we must define and quantify them. We define *risk* as a quantity of *severity* times a quantity of *likelihood*. We will identify risks and assign values for severity and likelihood to them (to the best of our ability). If a given risk is both severe and likely enough, we will attempt to minimize the risk, which involves reducing the likelihood of the event occurring, the severity of the consequences should it occur (harm reduction), or both. Risks for which we cannot identify any means of mitigation have been left out of the analysis.

Risks

After some discussion, we have identified the following risks:

- No multi-PSK provider willing to provide trial version/APs
- Prolonged absence due to disease or unforeseeable events
- Prolonged downtime in Uninett's network
- Loss of data
- Conflict/disagreement between the candidates
- Delays due to overly ambitious designs
- Hardware failure/complications

Severity and likelihood of events

Risk threshold

Given the likelihood p of a risk, the severity s of a risk, and a threshold t , the following equation outputs a value indicating whether preventative and harm reducing measures should be taken (1) or not (0):

$$f(p, s, t) = \begin{cases} 0 & ps < t \\ 1 & ps \geq t \end{cases}$$

The area of a chart that should be marked as "dangerous" is given with

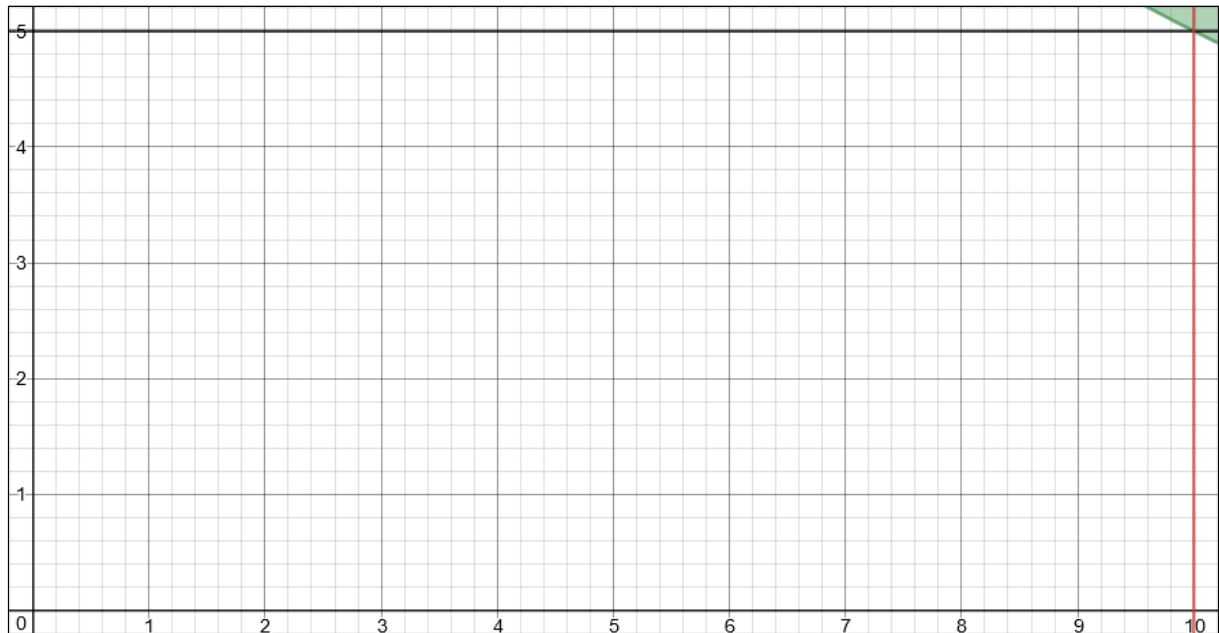
$$xy \geq t$$

where t is the threshold.

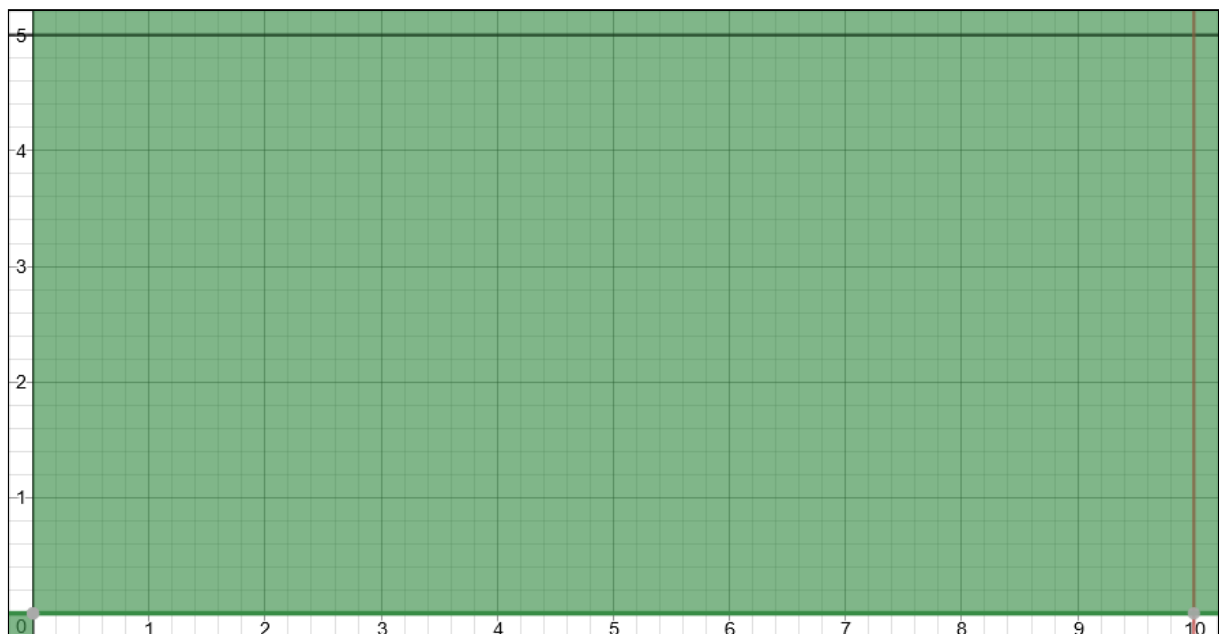
The choice of values for severity and likelihood of events involves a fair bit of guesswork unless there is historical data. In addition, how prone one is to a certain risk may vary depending on your behavioral traits, equipment and other factors. In our situation, most of

these values must be based on our own experience. To account for the possibility of optimistic values for likelihood and severity, we have also chosen a fairly low threshold value.

The threshold value must be set with the range of possible values for p and s . If p may range from 0 to 10, and s may range from 0 to 5, the maximum possible value for the threshold is ps , or 50:



The minimum possible value for the threshold is always 0 (*all risks should be dealt with*):

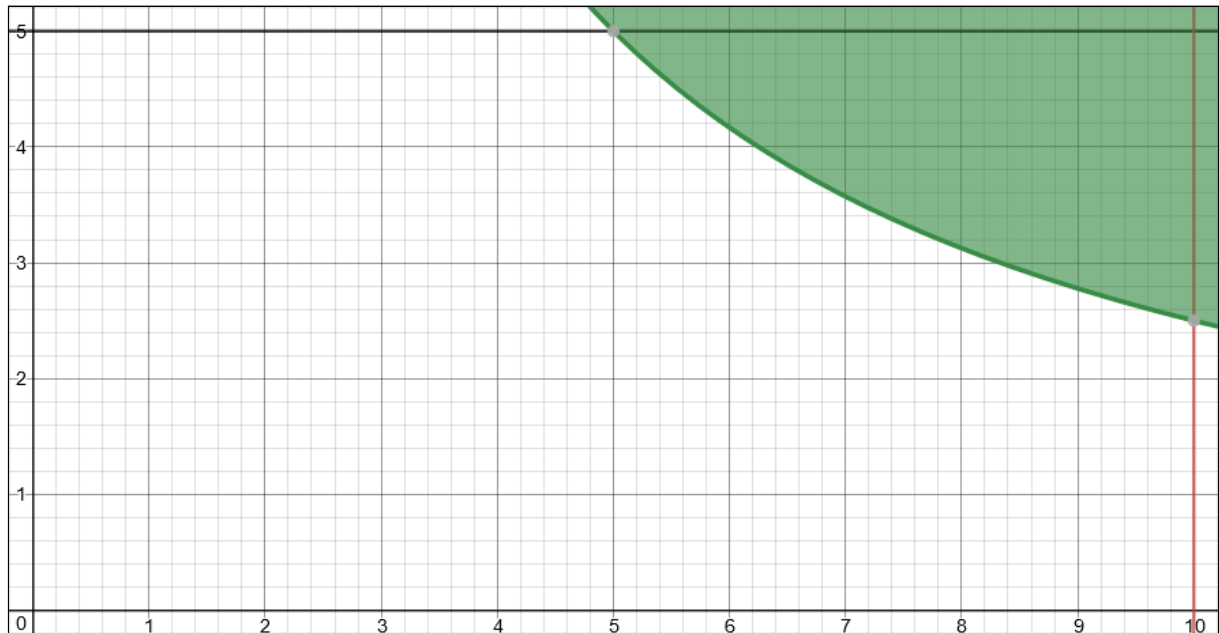


Because threshold value we choose depends on the number of levels for severity and likelihood, we define our threshold as a percentage of the $max(p)*max(s)$. We will call this value, which lies between 0 and 1, the *adjusted threshold*, as it adjusts for the number of levels of severity and likelihood. If we define six levels of severity (0 to 5) and 11 levels of

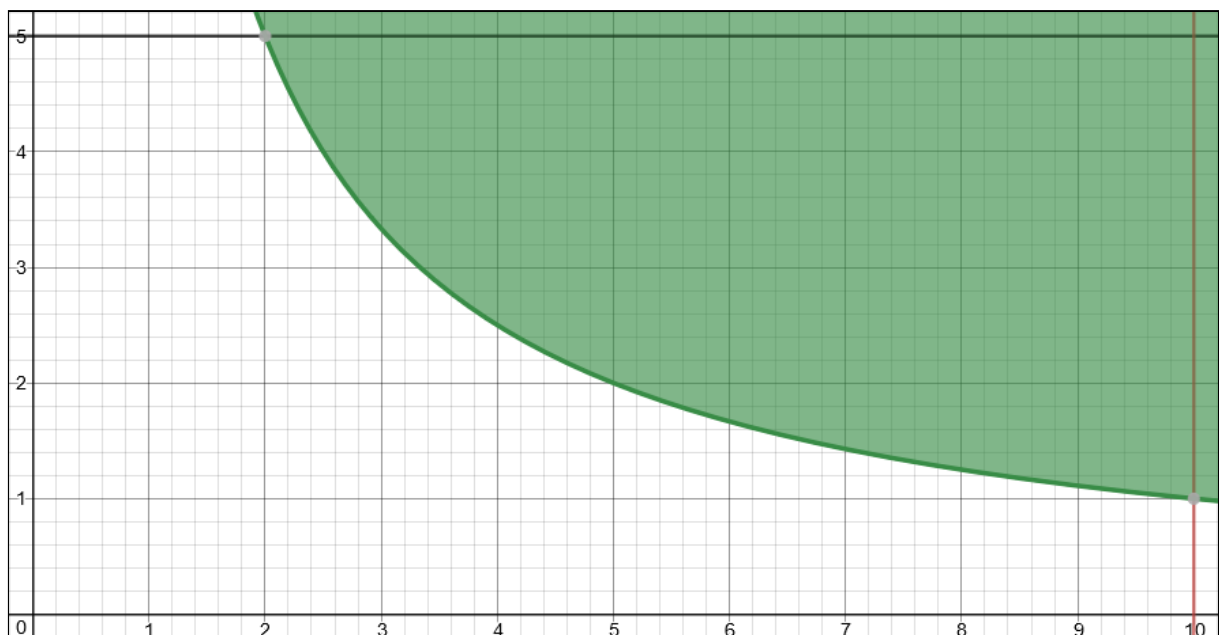
likelihood (0 to 10), then an adjusted threshold of 50% would be converted to the following non-adjusted threshold:

$$a = 0.5 \cdot 10 \cdot 5 = 25$$

In this example, if the likelihood times the severity of an event is greater than or equal to 25, measures should be taken to reduce the likelihood of the event occurring and/or to reduce the harm should the event occur.



We have chosen an adjusted threshold of 20%.



We consider this an appropriate degree of preparedness for this project, as it will cover both “thinkable” events with a high degree of severity, and highly likely events with little severity.

Definitions of levels

We define six levels of severity thusly:

Level	Name	Description
0	Negligible	Events whose resulting damage is clearly visible and can be corrected swiftly and easily
1	Nuisance inducing	Events that may require diagnosis and/or repetition of tasks
2	Delaying	Events that may cause minor delays or force us to reconsider our plans
3	Concerning	Events with the potential to cause significant delays and/or minor loss of work
4	Dangerous	Events with the potential to cause great delays and/or significant loss of work
5	Critical	Events that threaten to invalidate large portions of previous work, setting us back weeks and threatening the project

Furthermore, we define 11 levels of likelihood:

Level	Name	Description
0	Negligible	Events that almost certainly will not happen
1	Newsworthy	Events that would be newsworthy
2	Highly unlikely	Highly unexpected events
3	Improbable	Unexpected events
4	Thinkable	Events that would surprise us if they occurred
5	Plausible	Events that would mildly surprise us if they occurred
6	Probable	Events that would not surprise us if they occurred
7	Common	Events that happen often, or that should be expected given present circumstances
8	Highly likely	Events that will probably happen at least once
9	To be expected	Events that should be expected occasionally
10	Certain	Events that will almost certainly happen frequently

Estimation of severity and likelihood of risks

We will attempt to estimate the likelihood and severity of the identified risks.

1. No multi-PSK provider willing to provide trial version/APs

If we are to develop the “optimal” solution, least one of the providers of multi-PSK technology (namely Aerohive, Ruckus, Aruba and Riverbed) must be willing to lend us a compatible access point and give us access to any necessary software. We have inquired (at the time of writing, at least two weeks ago), but thus far, none has responded. Luckily, hostapd allows us to set one or more PSK for a specific MAC address. It also allows us to set one or more PSKs for “all” MAC addresses. This can be used to simulate these components, and we will still be able to develop a proof of concept. Therefore, the severity of this event is lowered significantly.

Based on these considerations, we give this risk a likelihood of 7 and a severity of 2.

2. Prolonged absence due to disease or unforeseeable events

Prolonged (>1 week) absence could cause delays. These delays would likely be significant, but not critical. During our five semesters at NTNU, we have experienced prolonged absence due to disease in our group on average approximately twice. These groups typically consist of four or more members. We have rarely experienced sickness that prevents us from doing work.

Based on these considerations, we give this risk a likelihood of 5 and a severity of 2.

3. Prolonged downtime in Uninett's network

We consider Uninett's systems robust; networks are their specialty. Furthermore, our development does not depend on their networks, as we will be using local area networks for testing.

Based on these considerations, we give this risk a likelihood of 2 and a severity of 1.

4. Loss of data

Primarily for practical purposes, we mainly use cloud services for storage of data. We also make sure we have multiple copies of any work. This gives us a high degree of protection against loss of data. Should data be lost, however, it would have the potential to set us back greatly.

Based on these considerations, we give this risk a likelihood of 2 and a severity of 4.

5. Conflict/disagreement between the candidates

We chose to work together on our dissertation because we are generally on the same page about all major decisions. Both candidates welcome sound argumentation for an opposing view, and are willing to change their position when presented with such sound arguments. We find the event of conflict about significant aspects of the solution highly unlikely. If such disagreements should occur, we would agree to let Uninett make the decision.

Based on these considerations, we give this risk a likelihood of 2 and a severity of 1.

6. Delays due to overly ambitious designs

We have a history of proposing overly ambitious designs and solutions which have caused us to present unpolished products in previous projects. The deadline is rigid, and we have a busy schedule ahead. Overly ambitious designs could, in the worst case, lead to failure.

Based on these considerations, we give this risk a likelihood of 7 and a severity of 4.

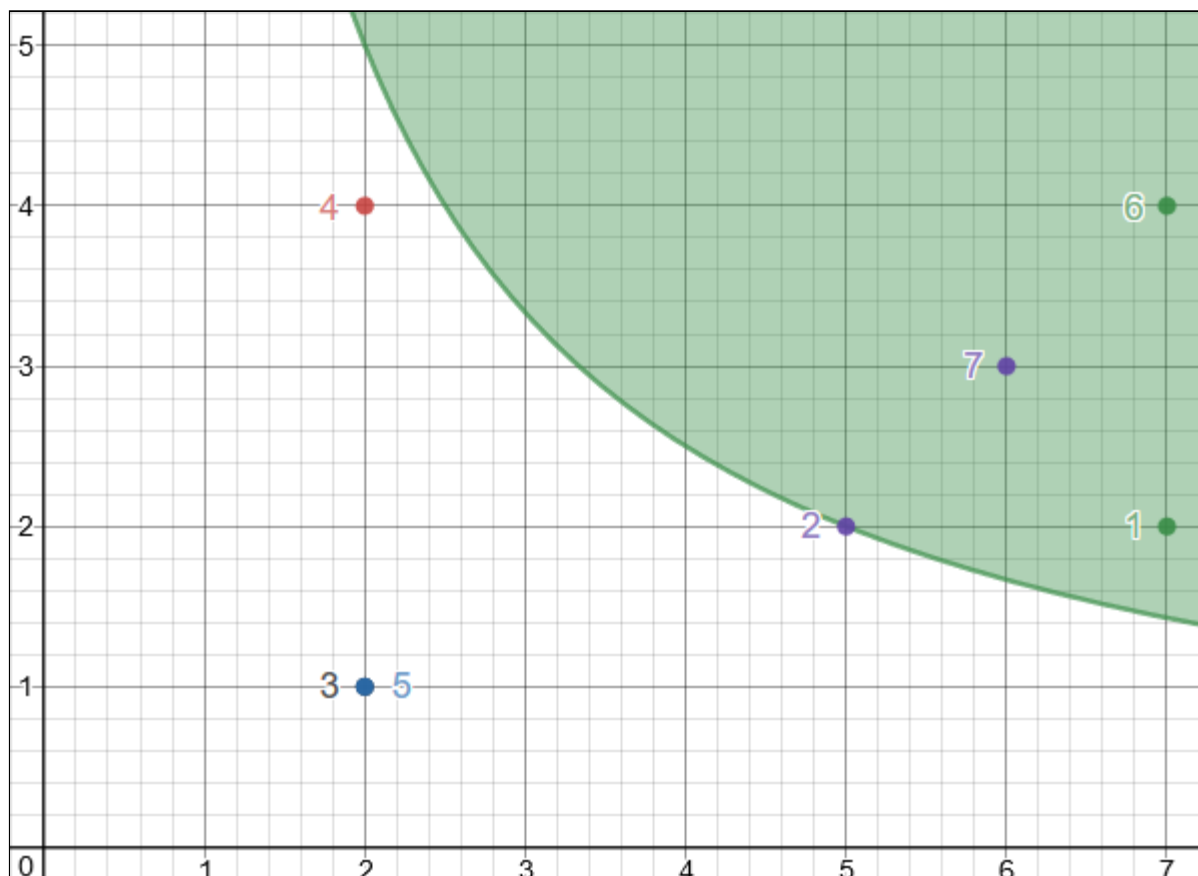
7. Hardware failure/complications

Our project relies on several pieces of hardware. Some of this hardware would be difficult to replace in a timely manner. We have experienced hardware issues with computers lately. If the computer in question breaks down, it could lead to significant delays. There is also a possibility of hardware incompatibilities. In that case, finding and acquiring compatible hardware may cause minor delays and extra expenses.

Based on these considerations, we give this risk a likelihood of 6 and a severity of 3.

Graphical representation

We have now assigned levels of severity and likelihood to all identified risks (for which there is potential for mitigation). Plotting in the coordinates of our risks yields the following:



1) No multi-PSK provider willing to provide trial version/APs, 2) Prolonged absence due to disease or unforeseeable events, 3) Prolonged downtime in Uninett's network, 4) Loss of data, 5) Conflict/disagreement between the candidates, 6) Delays due to overly ambitious designs, 7) Hardware failure/complications

Risk management

There are four risks that need management: 1) No multi-PSK provider willing to provide trial version/APs, 2) Prolonged absence due to disease or unforeseeable events, 6) Delays due to overly ambitious designs, and 7) Hardware failure/complications.

No multi-PSK provider willing to provide trial version/APs

In order to reduce the likelihood of rejection, we will ask Uninett to request a trial and necessary hardware on our behalf. Their organizational email address holds more weight than ours.

In order to reduce the severity of this event, we will begin planning to use hostapd instead. This way we will have a plan, even if we do not get access to proprietary technology, and we will be able to develop an approximation that can easily be modified to operate against proprietary technology instead.

Prolonged absence due to disease or unforeseeable events

Besides practicing good hygiene, the most efficient measure we can take in order to mitigate this risk is to reduce the severity by planning for some absence. We have attempted to exaggerate the time needed in each phase of the project period. Furthermore, we will implement a routine consisting of summarizing and explaining what we have worked on and what needs to be done at the end of every day, thus increasing the chances that the other candidate will be able to fill both roles for a limited period of time.

Delays due to overly ambitious designs

In order to reduce the likelihood of this event, we will develop minimalistic designs that implement only the strictly necessary functionality initially. If there is time, we may decide to improve these designs at a later time. A high degree of minimalism will be added as a quality control requirement.

Hardware failure/complications

Uninett is able to replace any hardware that fails except our computers. We will take care not to store any single copy of data on our computers that will cause delays if lost. Additionally, we will create a full system image backup for both computers, so that we can restore their state.

In order to reduce the likelihood of hardware failure, we will handle hardware with care and regularly tighten loose screws if we detect them by the sound they cause.

Analysis of financial feasibility

The two possible solutions are different financially. The cost of the “hotspot solution” depends on the number of users that should be able to connect IoT devices simultaneously, while the cost of the “multi-PSK solution” depends mainly on required coverage in a building. We will be somewhat pessimistic in our approach in order to avoid surprises.

Non-quantifiable benefits

Non-quantifiable benefits are benefits that cannot be measured with any reasonable degree of accuracy. We must assume that non-quantifiable benefits exist as there is a demand for IoT support, but no reason for this demand based on quantifiable benefits is given by Uninett. Some such benefits could potentially be made quantifiable if there was a list of offers made by universities for this functionality, for example, but to our knowledge, no such list exists at this point in time.

Non-quantifiable benefits could include the improved image of being on the cutting edge that is achieved by being IoT-compatible. It could also include the improved opportunities for research, convenience and automation. We are operating with rough estimates in this analysis, and the deduced figures should be assumed to be wildly inaccurate given our lack of definite data.

Financial feasibility of the hotspot solution

The hotspot solution comes with a significant increase in administrative work. This work is related to the configuration, distribution and management of hotspot devices. We will proceed with the assumption that one employee can manage approximately 500 hotspot devices, and that each employee receives 400,000 NOK in annual wages. Furthermore, we will operate with an expected lifespan of 3 years. We expect approximately 10% of existing devices to break or disappear each year. The current lowest cost of a Raspberry Pi is 269 NOK² and the cost of the compatible Asus USB-N10 Wi-Fi adapter is 114.40 NOK³, both excluding taxes and assuming there is no possibility of wholesale discounts. It is difficult to predict the benefits of the systems, but we expect these benefits to diminish rapidly as standards are implemented for IoT devices and new technology emerges, rendering this solution obsolete. We estimate a halving of non-quantifiable benefit each year. We are operating with a discount factor of 10%.

² From Dustin:

https://www.dustin.no/product/5010909893/3-model-b-12ghz-64-bit-arm-1gb-ram-wifibt?ssel=false&utm_campaign=prisjakt&utm_source=prisjakt.no&utm_medium=pricecompare&utm_content=5637175076. Accessed on February 17, 2019.

³ From Multicom:

<https://www.multicom.no/asus-usb-n10-nano-nettverksadapter-usb/cat-p/c/p7026696>. Accessed on February 17, 2019.

If we decide that at most 500 users should be able to connect IoT devices simultaneously, the below figure shows the approximate cost and benefit:

Cost	Year					
	0	1	2	3	4	5
Training	30000	10000	0	0	0	0
Hardware	191700	9585	9585	172530	18211,5	18211,5
Support/administration	400000	400000	400000	400000	400000	400000
Sum cost	621700,000	419585,000	409585,000	572530,000	418211,500	418211,500
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of costs	621700	381441	338500	430150	285644	259676
Cumulative present value of cost	621700	1003141	1341641	1771791	2057435	2317112
Benefit						
Quantifiable benefit	0	0	0	0	0	0
Non-quantifiable benefit	500000	250000	125000	62500	31250	15625
Sum benefits	500000	250000	125000	62500	31250	15625
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of benefit	500000	227273	103306	46957	21344	9702
Sum gain	-121 700 kr	-154 168 kr	-235 194 kr	-383 193 kr	-264 300 kr	-249 975 kr
Cumulative present value	500 000 kr	727273	830579	877536	898880	908582
Cumulative gain + cost	-121 700 kr	-275 868 kr	-511 062 kr	-894 255 kr	-1 158 555 kr	-1 408 530 kr

If we decide that only about 50 users should be able to connect IoT devices simultaneously (for research, for example), the results are different. We assume that existing staff will be able to handle this additional administrative workload, and that these 50 users provide more benefit than the average user. We will halve the non-quantifiable benefits.

Cost	Year					
	0	1	2	3	4	5
Training	30000	10000	0	0	0	0
Hardware	19170	958,5	958,5	17253	1821,15	1821,15
Support/administration	0	0	0	0	0	0
Sum cost	49170,000	10958,500	958,500	17253,000	1821,150	1821,150
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of costs	49170	9962	792	12962	1244	1131
Cumulative present value of cost	49170	59132	59924	72887	74131	75262
Benefit						
Quantifiable benefit	0	0	0	0	0	0
Non-quantifiable benefit	500000	250000	125000	62500	31250	15625
Sum benefits	500000	250000	125000	62500	31250	15625
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of benefit	500000	227273	103306	46957	21344	9702
Sum gain	450 830 kr	217 310 kr	102 514 kr	33 995 kr	20 100 kr	8 571 kr
Cumulative present value	500 000 kr	727273	830579	877536	898880	908582
Cumulative gain + cost	450 830 kr	668 140 kr	770 654 kr	804 649 kr	824 749 kr	833 320 kr

Conclusion

The hotspot solution is probably not financially feasible for large scale deployment. For small-scale research operations and similar, however, the costs are drastically reduced, and much of the benefit is preserved. Our conclusion is that the hotspot solution is worthwhile to implement for small-scale research operations (around 50 simultaneous users maximum) if the multi-PSK solution cannot be implemented.

Financial feasibility of the multi-PSK solution

The most significant factor in the financial feasibility of the multi-PSK solution is the coverage of IoT connectivity, which decides how many access points need to be installed or upgraded. In order to be equally pessimistic about this solution, which we are otherwise fairly optimistic about, we will assume that old access points have lost their entire market value and cannot be sold in order to cover some of the cost. We will, however, assume that this solution has longevity. It is theoretically a perfectly secure and efficient solution with minimal administrative work required. The lifespan of an access point is somewhat pessimistically assumed to be five years. We have chosen to use Aerohive’s prices in this analysis for two reasons: 1) Aerohive has recently responded to our request to borrow access points and a license key (see the *Amendments* section), and 2) Aerohive is used by Surfnet, the “Uninett of the Netherlands.”

If our understanding is correct, all new Aerohive access points are compatible with their PPSK (Aerohive’s name for their multi-PSK technology) technology. Sadly, Aerohive uses a “call for pricing” model for the pricing of licenses. The price of a license also depends on the number of access points to be supported and the duration of the license. Perpetual licenses are also available. Because we cannot find any definite prices, we will have to make estimations.

Product	Price in NOK excluding taxes
Access point	1904 NOK ⁴
License for up to 2000 APs	15537 NOK ⁵
Additional (pessimism factor): Support and renewal for 5 years, etc.	15000 NOK

We assume that very little training is required. Administrators only need to know how to operate a simple dashboard and possibly manually terminate sessions. Furthermore, we assume that the non-quantifiable benefits decline only slightly with time, as we expect this solution to be much more in line with future solutions than the hotspot solution. We believe the solution will age well.

If we assume that 500 rooms should allow connection of IoT devices (and each room requires its own access point), the following estimates apply:

⁴ From Atea’s e-shop: <https://www.atea.no/eshop/product/dell-emc-networking-aerohive-ap130/?prodid=3880978>. Accessed on February 18, 2019.

⁵ “HiveManager NG Virtual Appliance can be easily deployed on a customer’s existing server infrastructure at a total cost of \$1799.” Aerohive press release from June 22, 2017. Available at: https://www.aerohive.com/press_releases/aerohive-delivers-the-power-and-simplicity-of-cloud-networking-to-on-premises-environments/. Accessed on February 18, 2019.

Cost	Year					
	0	1	2	3	4	5
Training	10000	0	0	0	0	0
Hardware	952000	0	0	0	0	0
Licenses and fees	15537	0	0	0	0	0
Support/administration	15000	0	0	0	0	0
Sum cost	992537,000	0,000	0,000	0,000	0,000	0,000
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of costs	992537	0	0	0	0	0
Cumulative present value of cost	992537	992537	992537	992537	992537	992537
Benefit						
Quantifiable benefit	0	0	0	0	0	0
Non-quantifiable benefit	500000	450000	405000	364500	328050	295245
Sum benefits	500000	450000	405000	364500	328050	295245
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of benefit	500000	409091	334711	273854	224063	183324
Sum gain	-492 537 kr	409 091 kr	334 711 kr	273 854 kr	224 063 kr	183 324 kr
Cumulative present value	500 000 kr	909091	1243802	1517656	1741718	1925042
Cumulative gain + cost	-492 537 kr	-83 446 kr	251 265 kr	525 119 kr	749 181 kr	932 505 kr

Increasing this to 2000 rooms results in larger initial losses, but also larger cumulative benefits after a five year period. While we have multiplied the hardware costs by 4, we have only multiplied the non-quantifiable benefits by 3.

Cost	Year					
	0	1	2	3	4	5
Training	10000	0	0	0	0	0
Hardware	3808000	0	0	0	0	0
Licenses and fees	15537	0	0	0	0	0
Support/administration	15000	0	0	0	0	0
Sum cost	3848537,000	0,000	0,000	0,000	0,000	0,000
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of costs	3848537	0	0	0	0	0
Cumulative present value of cost	3848537	3848537	3848537	3848537	3848537	3848537
Benefit						
Quantifiable benefit	0	0	0	0	0	0
Non-quantifiable benefit	1500000	1350000	1215000	1093500	984150	885735
Sum benefits	1500000	1350000	1215000	1093500	984150	885735
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of benefit	1500000	1227273	1004132	821563	672188	549972
Sum gain	-2 348 537 kr	1 227 273 kr	1 004 132 kr	821 563 kr	672 188 kr	549 972 kr
Cumulative present value	1 500 000 kr	2727273	3731405	4552968	5225155	5775127
Cumulative gain + cost	-2 348 537 kr	-1 121 264 kr	-117 132 kr	704 431 kr	1 376 618 kr	1 926 590 kr

Another thing to consider is the possibility of existing access points supporting multi-PSK technology in the near future. In that case, the hardware costs would be non-existent, and the following estimate is given:

Cost	Year					
	0	1	2	3	4	5
Training	10000	0	0	0	0	0
Hardware	0	0	0	0	0	0
Licenses and fees	15537	0	0	0	0	0
Support/administration	15000	0	0	0	0	0
Sum cost	40537,000	0,000	0,000	0,000	0,000	0,000
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of costs	40537	0	0	0	0	0
Cumulative present value of cost	40537	40537	40537	40537	40537	40537
Benefit						
Quantifiable benefit	0	0	0	0	0	0
Non-quantifiable benefit	1500000	1350000	1215000	1093500	984150	885735
Sum benefits	1500000	1350000	1215000	1093500	984150	885735
Discount rate = 15%	1,000	0,909	0,826	0,751	0,683	0,621
Present value of benefit	1500000	1227273	1004132	821563	672188	549972
Sum gain	1 459 463 kr	1 227 273 kr	1 004 132 kr	821 563 kr	672 188 kr	549 972 kr
Cumulative present value	1 500 000 kr	2727273	3731405	4552968	5225155	5775127
Cumulative gain + cost	1 459 463 kr	2 686 736 kr	3 690 868 kr	4 512 431 kr	5 184 618 kr	5 734 590 kr

Conclusion

The multi-PSK solution seems financially viable, and this is by far the most reasonable solution both practically and financially, regardless of which of the three presented scenarios apply. While it is a significant investment, this is approximately what large-scale IoT support support costs today.

The spreadsheets are attached.

Guidelines and standards

In order to ensure the quality of our work, that our work is understood, and that the needs of Uninett are met to the fullest degree possible, we have established the following guidelines and standards.

Documentation requirements

Both candidates shall keep an individual timesheet, which shall state how many hours were spent working on the project each day (unless zero hours were spent). A brief description of the activities that were undertaken on the day shall accompany each entry. The number of hours spent on the project each week shall be summarized and presented in a sensible format.

A progress plan for the entire project period shall be delivered as soon as possible in a shared folder to which the stakeholders have access.

Every Friday (or on Monday the following week at the latest), a progress plan for the coming week shall be developed and placed in the shared folder. The progress plan shall state which tasks are planned, show which tasks are planned for which days of the week, and indicate which days will be spent in Uninett's offices, and which will be spent elsewhere.

Every Friday (or on Monday the following week at the latest), a weekly report outlining the activities of the week that has passed shall be delivered to the shared folder. The weekly report shall give the reader insight into the candidates' progress, problems and plans.

When meetings are held, one of the candidates shall be given the responsibility of writing minutes of the meeting. The minutes of the previous meeting shall be approved by the attenders. The minutes shall at a minimum include an agenda, which shall be published in the shared folder prior to the meeting, as well as a summary of what was expressed at the meeting, by whom. The minutes of the meeting shall be published in the shared folder as soon as possible after the meeting.

A prestudy (this document) shall be prepared. The prestudy shall outline the candidates' research on the topic, define goals, specify the scope and plans for progress, communication, project organization, standards, guidelines and success criteria, analyze the stakeholders, external conditions, relevant regulatory frameworks, the financial feasibility of the project, risks (risk analysis and management plan), identify functional and non-functional requirements of the solution, and list all sources used in a bibliography.

A design report shall be prepared. This report shall include complete designs, including user experience, information flow, graphical design, functionality, and other specifications of all essential parts of the solution to be developed, as well as give an account of the design process.

Additionally, an execution report shall be prepared. This report shall explain the technical details of how the solution is design, including any code, hardware configuration and

software used, and give an account of the execution process. It shall also include explanations of how the systems are tested, the results of said testing, any improvements or changes made, how the system is deployed and other relevant information.

Finally, a final report shall be prepared, which contains all the three reports described above, as well as a preface and conclusion.

The following documents shall be written in English:

- The weekly reports
- The prestudy report (this document)
- The design report
- The execution report
- The final report
- Any documents attached to these documents

We consider these documents the most important ones, and their being written in English ensures that they can be understood by the majority of potential readers. Other documents may be written in Norwegian.

The following documents should preferably be formatted using LaTeX:

- The prestudy report (this document)
- The design report
- The execution report
- The final report

Quality control requirements

The prestudy report, the design report, the execution report and the final report should be reviewed by Uninett, the NTNU representative (assistant professor Stein Meisingseth) and the candidates before finalization and revised according to feedback. The candidates should proofread the documents before submission.

When a document is revised, all other documents that may reference the revised document should be checked, and the validity of the references should be verified. For example, if a new illustration is added to the prestudy report, the design report and execution report should be checked for references to illustrations in the prestudy report. These references should be updated if they are no longer valid.

Unit tests should be written for code, and these tests should have decent coverage. When a bug is discovered and fixed, a unit tests should be written that attempt to trigger that specific bug. Integration test routines should also be defined and run regularly. Components that fail these tests are not to be considered complete and must be completed before the product is finalized.

In addition to tests, a focus group of at least three people should be asked to test the systems. If they are unsuccessful in using the systems, they are not to be considered

complete. Any constructive feedback given during these trials should be implemented if possible.

Controls and methodology

Methodology

We will use a Scrum style framework in our design and development process. Issues (epics, stories, tasks and bugs) will be defined as they are identified and assigned in sprints. Issues will fall under one of four stages: 1) open, 2) in progress, 3) review, and 4) done. These issues may be tracked through Jira or with sticky notes, for example. There will be no Scrum Master or Product Owner, however. The candidates share the responsibilities of the former, and the candidates share the responsibilities of the latter with Uninett. Every day will begin with a meeting that summarizes, prioritizes, assigns and identifies new issues.

Standards and principles

We wish to provide a platform-independent solution, meaning the solution should not in principle care which operating system it is running on. This means that our code should be highly modular, and platform- or product-specific code should be contained within an easily replaceable module for improved compatibility and customizability. Furthermore, a platform-independent programming language, such as Python, must be used, and components and libraries depending on specific platforms should be avoided. Code should also follow the “don’t repeat yourself” principle. We will strive for a high standard of code and design.

Our code should be easy to interpret, and its security should be demonstrated, through extensive use of comments and flowcharts.

Controls

The following controls shall be implemented:

- The proper encryption of packets shall be verified using packet sniffers and documented.
- It shall be confirmed that the design is as minimal as possible while still providing required functionality.
- It shall be confirmed that every iteration of the product is developed according to the designs.

Project organization

This project is organized with the following roles:

Role	Person(s)
Advisor (NTNU)	Stein Meisingseth
Advisor (Uninett)	Jørn Åne de Jong
Project managers	Jørn Åne de Jong, Tom Ivar Myren, Otto Wittner
Sponsor	Uninett AS
Document manager	Magnus Bakke
Developers/candidates	Magnus Bakke & Liang Zhu

Work on this project will primarily take place in Uninett's offices. Additional work will take place at the university and elsewhere.

The finished reports, code and other documentation will be submitted to the university at the end of the project period.

Conclusion

The technology required to securely connect IoT devices that do not support authentication using certificates exists, but it is fairly new and not extensively tested. Deploying either of the two possible solutions at scale is a risk but, in the absence of IoT standards, it is a necessary one if we are to be ready for the future of networking.

We are confident that we will be able to develop an elegant solution with a satisfactory level of security. Security should be the primary concern in uncharted waters, and this is achieved through proper encryption. The encryption standards used in modern wireless networks are strong. The issue lies in the dependence on trust. We cannot simply transfer a trust-based home network architecture to an enterprise environment. Therefore, we must extend this architecture and turn it into a trustless system. The relatively simple invention of user-specific pre-shared keys enables us to do just this. This is why we will focus our efforts on developing a highly automated system based on multi-PSK technology in which users can freely and securely connect any device that does not support certificate authentication and encryption. Everything points to this being a great investment that will open new possibilities for automation, convenience and research. The time is ripe.

With our thorough research of the topic and preliminary sketches for designs, we are now looking forward to moving on to the design phase, in which we will plan our solution in great detail.

Amendments

- February 11, 2019** Aerohive responded positively to our request. This means that we will almost certainly be able to design and execute the multi-PSK solution. The risk of no multi-PSK technology provider being willing to lend us access points and licenses has been virtually eliminated.
- February 20, 2019** Clarification: We have been using the terms “eduroam credentials” and “Feide credentials” interchangeably because students will have the same credentials on both services. We made the distinction clearer, and better explained that the multi-PSK solution does not make devices compatible with eduroam.
- March 25, 2019** Specified the cases in which it is necessary to have an “authentication dashboard” with a field for the user’s password.
- May 14, 2019** We found a new, lower price for the AP122 access point at 1195 NOK (tax-exclusive): <https://www.dustin.no/product/5011072112/>. This makes the multi-PSK solution even more financially feasible.

Bibliography

de Ridder, S. (2017, June 8). How can you connect your devices safely and simply to your WiFi network? - SURF Blog [Blog post]. Retrieved February 4, 2019, from <https://blog.surf.nl/en/how-can-you-connect-your-devices-safely-and-simply-to-your-wifi-network/>

hostapd configuration file [Configuration file]. (n.d.). Retrieved February 10, 2019, from <https://w1.fi/cgiit/hostap/plain/hostapd/hostapd.conf>. Read: *WPA/IEEE 802.11i configuration*.

Norwegian legal system. (2018a, June 15). Lov om behandling av personopplysninger (personopplysningsloven) [directive]. Retrieved February 17, 2019, from <https://lovdata.no/dokument/NL/lov/2018-06-15-38>

Norwegian legal system. (2018b, July 2). Forskrift om arbeidsgivers innsyn i e-postkasse og annet elektronisk lagret materiale [directive]. Retrieved February 17, 2019, from <https://lovdata.no/dokument/SF/forskrift/2018-07-02-1108>

Norwegian legal system. (2018c, June 15). Forskrift om behandling av personopplysninger [directive]. Retrieved February 17, 2019, from <https://lovdata.no/dokument/SF/forskrift/2018-06-15-876>

GÉANT Association. (n.d.). Disclaimer – eduroam [disclaimer]. Retrieved February 17, 2019, from <https://www.eduroam.org/disclaimer/>

Atlassian. (n.d.). Jira | Issue & Project Tracking Software | Atlassian. Retrieved February 17, 2019, from <https://www.atlassian.com/software/jira>

Peters, C. (2012, September 7). Code 3 Key Software Principles You Must Understand [explanation of the ‘Don’t Repeat Yourself’ principle]. Retrieved February 17, 2019, from <https://code.tutsplus.com/tutorials/3-key-software-principles-you-must-understand--net-25161>

AUTHENTICATION IN THE INTERNET OF THINGS

Design report

Part of a Bachelor's thesis

**Presented to the Institute of computer technology and informatics
of the Norwegian University of Science and Technology
by Magnus Bakke & Liang Zhu**

Submitted in partial fulfillment for Bachelor's degree of
Informatics with specialization in network administration
during the year 2016–2019

Introduction

We, the candidates, are two students at the Norwegian University of Science and Technology, where we study Informatics with specialization in network administration. In connection with our Bachelor's thesis, Uninett AS — the state-owned company responsible for Norway's network infrastructure in research and education — has requested that we research and develop a solution for getting IoT devices connected to the internet in a secure manner.

The project began with extensive research and planning, called the research stage. The result of this stage is outlined in the prestudy report, which precedes this document. This document describes, in detail, the result of the *design stage*, in which we make use of our research in the design of a solution before these designs are actualized in the coming execution stage.

Summary of requirements

A typical home has a wireless network secured using a password. In order to use the wireless network, the user must possess the secret key (called the pre-shared key). This key is used to encrypt and decrypt communications, and the ability to encrypt and decrypt communication is used as authentication. When the user is authenticated by means of possessing the secret key, they are authorized to use the network. This type of wireless network is called a *personal* wireless network. Its security is based on trust and secrecy: The password is shared only with people trusted by the network's owner, and is otherwise kept secret. Personal networks use the *WPA2-Personal* mode of the WPA2 encryption standard.

In an *enterprise* network with potentially hundreds of users, this system of trust and secrecy collapses. A network secured with a key known to hundreds of people cannot be considered secure at all. In addition, it is typically not desirable that one user of the wireless network is able to monitor the communications of other users, which is very much the case in today's pre-WPA3¹ personal networks. Therefore, enterprise networks are secured using the *WPA2-Enterprise* mode of the WPA2 standard, in which each user is given their own unique identity, and communication is encrypted in a way that prevents other users from decrypting it. This is achieved using digital certificates, an application of public-key cryptography. In short, there is no master password; there are identities that cannot be forged, as they are mathematically secured.

With the advent of the internet of things (IoT), in which small devices (often lacking user interfaces and typically serving a singular, highly specific purpose such as measuring and reporting on the temperature or humidity in a space), issues arise. IoT devices are typically marketed to consumers for use in so-called smart houses. They are designed for personal networks, and are not equipped with the functionality required for authentication in enterprise networks. The world of *master passwords* is the only one they know.

¹ Special consideration is given to the benefits of WPA3 versus WPA2 in the execution report.

Still, there is a demand for IoT devices in enterprise networks as well. The question is: How can we connect IoT devices to an enterprise network in a way that can be considered secure? We have hypothesized two possible solutions, one of which has been selected after careful consideration.

Introduction to chosen of solution

The *hotspot solution*, which was always considered a backup solution, was based on the idea of configuring small boxes containing computers without displays to act as hotspots for IoT devices. The devices would be leased to users upon request, and the user would be able to see a dashboard by connecting to the device's IP address in a web browser. After the user has inputted their eduroam credentials, the device would broadcast a hotspot with a dynamically generated, strong and random PSK. The hotspot acts as a bridge between the IoT device and eduroam.

This solution has several problems: Firstly, it comes with a lot of additional administrative work: The hotspot devices must be reset to their "ready" state, distributed to users upon request, and kept track of. Such hardware is prone to malfunction, wear and tear, and there is no simple way of supporting roaming between hotspots (one hotspot belongs to one user), meaning that the hotspot device must be carried around so that they are always nearby if the connected devices are moved. This requires that the hotspot devices are battery-powered and always remain charged. Lastly, the hotspots would either cause harmful interference with existing wireless networks or drastically reduce the number of channels that existing access points may be configured to use. We found that the solution is financially feasible if it is only used by a small number of people, such as a few research teams, in isolated areas.

The *multi-PSK solution* takes a different approach entirely. Some modern access points support multiple PSKs by virtue of advanced controller software. Given the right functionality, this means that a user may be able to use a customized web dashboard to request a new PSK for their devices. The PSK would be generated, and devices would be able to communicate with wireless access points normally using the newly generated PSK. We have found four hardware manufacturers that have implemented multi-PSK technology in some form. These are Aerohive, Ruckus, Aruba (coming soon, according to representatives), and Riverbed. There is also limited support in hostapd. We have requested access points and software licenses from all four of these companies.

Aruba responded positively, but multi-PSK support is planned for the next version of Aruba's ClearPass at the time of writing. The release date of this next version is not known, and we cannot risk waiting for it. Luckily, Aerohive has also responded positively, and we are now in possession of two of their access points, which were generously sent to us by Aerohive. We would like to thank both Aruba and Aerohive for taking an interest in our project. We have verified that HiveManager — Aerohive's cloud networking solution — allows us to develop a custom dashboard, thanks to the support for automation using its API. We will refer to the "multi-PSK solution" as the "PPSK solution" from now on. Private Pre-Shared Key (PPSK) is Aerohive's name for the technology. This is our chosen solution.

Reasoning for the choice of solution

By generating a unique PSK for each IoT device to be connected, we have corrected the inherent incompatibility between IoT devices and enterprise networks, namely the reliance on *master passwords*. The solution does not require any major changes to network architecture, and problems with roaming, portability and harmful interference are also non-existent. In the prestudy, we found that this solution was indeed financially feasible.

Scope of the document

This document deals only with the theoretical design of the chosen solution. A combination of drawings, charts, mockups, textual descriptions, and logical reasoning constitute the designs of the solution. The designs we develop will not be executed (implemented in a working prototype) in this stage. Instead, we describe the system we intend to develop with a degree of detail that reduces the development itself to a relatively manual task consisting of writing the code, configuring hardware according to designs, testing and similar processes. This is the goal of the document.

Contents

Introduction	2
Summary of requirements	2
Introduction to chosen of solution	3
Reasoning for the choice of solution	4
Scope of the document	4
Contents	5
Method and approach to design	7
Principles in design	7
Detailed designs & descriptions	9
Hardware	9
Software and frameworks	9
Programming and markup languages	9
Operating systems	9
Architecture	10
Frontend	10
Receiver	10
Authenticator	10
Ad hoc adapter	10
Illustration	11
OAuth 2.0 Authentication	12
User interfaces	12
Form validation	13
Backend code components	13
Basic explanation of Django REST framework	13
Introduction to types and concepts in RESTful APIs and Django REST framework	14
Description of components and architecture	15
Class diagram	18
Reasoning	18
Data templates	19
Frontend details and code components	19
Success and error handling	20
Flow	22
Database	24
Security measures	24
Expiration of old pre-shared keys	24
Generation of strong pre-shared keys	25
Obscurity	25

SSL	25
Hiding of secrets from Git repositories	26
Version control with Git	26
Testing	26
Summary	27
Deviations from the prestudy report	27
Conclusion	27
Amendments	28
Bibliography	29

Method and approach to design

As required by our own recommendations in the prestudy report, we used a customized version of Scrum during the design process. In order to minimize slack, one candidate designed a component of the solution while the other candidate made prototypes of the same component. In other words, we have written preliminary drafts for the execution report in parallel with the writing of this document. The development of the prototype components has also affected the chosen design.

Principles in design

We choose to focus on five principles when designing our solution: 1) Reusability, 2) modularity, 3) platform independence, 4) minimalism, and 5) the *Don't repeat yourself* (DRY) principle. Note that these principles are closely related and complement each other in our case.

Reusability

As the term implies, reusability is measure of how readily and easily our solution and code components can be reused in other situations. Naturally, we are developing a system for getting IoT devices online and do not anticipate the solution being useful for other purposes. Still, we want the solution to be just as valid for an institution in another part of the world as it is for Uninett. Likewise, we wish to structure our code and components in such a way that they can be used wherever the same or highly similar functionality is required.

Modularity

Modularity is the degree of the independence of each module from one another. In other words, modular code is divided into modules that know very little about each other, and therefore do not contain dependencies pertaining to the specific nature of other modules. It is reasonable of one module to expect that another module has the required superficial *interface* functions, but nothing more specific should be assumed.

In our case, for example, the authentication module will have a function that accepts credentials, or any other data that is required by the specific solution, and returns a Boolean value indicating whether or not authentication passed (or raises an exception if authentication fails). This will be true in any environment. The authentication module will make it known what information is required. The receiver module will not, however, assume how this information is processed or anything else pertaining to the inner workings of the authentication module. These modules are explained in the *Architecture* subchapter of the *Detailed designs & descriptions* chapter.

A high degree of modularity allows us to customize each module to our needs while feeling confident that other components of the solution will not break as a result. This also makes the code easier to test and refactor.

Platform independence

Platform independence is a property of a solution that does not make assumptions about the

operating system it is running on or other variables in its environment, such as hardware models. We do not want our solution to depend on a certain operating system, nor do we want it to work only against the access points of a certain manufacturer.

Independence from the operating system is achieved by using a platform independent programming language and framework (Python and Django REST framework in our case), while independence from the manufacturer is achieved by the customizability of the *ad hoc adapter* (as described in the *Architecture* subchapter). Independence from specific Identity Providers such as Feide, and indeed from specific approaches to authentication such as OAuth 2.0 or SAML 2.0, is similarly achieved through customizability of the *authentication* module (also described in the *Architecture* subchapter).

Minimalism

In the prestudy, we identified the risk of overly ambitious designs: We have a history of planning too ambitiously within given framework conditions, such as deadlines. We will mitigate this risk by designing a minimal solution.

This involves leaving out functionality that is not strictly necessary, and instead plan for future expansion. The solution will form a basis for further development. Options for future development will be outlined in the execution report. We will attempt to maintain a high degree of elegance in our code and architecture, but will not risk missing deadlines for the sake of achieving this. Still, elegance is usually the simplest and fastest way to completion of a project.

This principle also applies to user interfaces, which should only feature fields and options that are strictly necessary. There will be no screen requiring that the user accepts terms of use, for example. This is an example of a feature that may be required of a full-scale deployment. Furthermore, we will not develop a dashboard for administration of registered users. This is a component that a full-scale deployment should feature, and while it is within our grasp, we estimate that there will not be sufficient time. The steps required to develop it will be outlined in the execution report.

Don't repeat yourself

Don't repeat yourself (DRY) is a principle of software design that is meant to reduce both the quantity of code required and the occurrence of repetition. This helps to reduce the complexity of the software solution and increase its maintainability. Following this principle involves identifying where similar functionality is required and refactoring it so that the functionality is only declared once, but available in both or all places where it is needed. This eliminates the need to maintain two identical or nearly identical components when fixing bugs, for example. The Django REST framework already follows this principle to a high degree. It is our responsibility to make sure the principle is followed throughout our own code as well.

Detailed designs & descriptions

This chapter includes designs for all planned aspects of the solution. We have attempted to order the descriptions of the solution's components in a way that minimizes the need to know details not yet described.

Hardware

Our solution requires at least one access point that supports PPSK. Aerohive has generously sent us two, which also allows us to verify that one PSK allows us to connect to different APs, and that roaming works as expected.

The controller software (HiveManager) will run in “the cloud” (on Aerohive's servers), as we do not have a plan that allows us to download a HiveOS image. Therefore, we do not need any hardware for this purpose. Our own servers will be virtualized, so we do not require any additional hardware for this either.

Software and frameworks

Our backend will be built using Django REST framework on account of its versatility and security. Django REST framework prevents us from having to “reinvent the wheel” — it allows us to program API endpoints belonging to certain URLs in *views*. With a custom view class, we will be able to program our receiver component (as is described initially in the *Architecture* subchapter) with minimal code repetition. Furthermore, it has seamless SQL integration that performs a lot of optimizations for us and makes database queries more readable.

Programming and markup languages

Django REST framework is a framework for backends written in Python, and the entirety of our backend will be written in Python. Our frontend will consist of HTML5 and CSS3, with JavaScript populating the user interface, sending requests and receiving responses asynchronously. We have planned for the frontend to be entirely customizable, and our solution will not make assumptions about it or its constituents, nor will it serve the frontend using templates.

Operating systems

We want our solution to be free, including the specifics of the environment. This means that we must not rely on an operating system that requires a paid license. Therefore, our solution will run on a Linux distribution, and our code will be written in Python, as stated. Python is one of many open programming languages, but it is one that we have experience with, and it is compatible with our choice of software.

We will not specify operating systems further, as we may find that one Linux distribution works better than others for our purposes. This exploration is a task for the execution stage.

Architecture

The solution will consist of four main components: 1) The frontend, 2) the receiver, 3) the authenticator, and 4) the ad hoc adapter. The latter three are components of the backend. These components are described below.

Frontend

The frontend will be a page reachable from within the network that features a form with fields for the required information, such as options for how the PSK should be delivered. The information provided by the user is sent to the receiver (over HTTPS) for validation. If the validation passes, the newly generated PSK is displayed to the user.

Receiver

The receiver receives PSK requests and credentials, prompts the authenticator to validate the credentials (which may consist of a username and password, some other secret, or nothing at all, depending on the implementation), and forwards the PSK request to the ad hoc adapter if authentication passes. This component is not replaceable, and the logic performed by the receiver should in principle be the same in all situations.

Authenticator

We want to build a solution that does not depend on Feide. In other words, we want this system to be compatible with any Identity Provider. Because different systems may use different authentication methods, we need a replaceable authentication module. In our case, the user must first use Feide's single sign-on (SSO), and the authenticator will simply verify that the user is authenticated. In another scenario, the authenticator may not perform any logic (if there is no authentication requirement, for example), but return *true* no matter the case. Custom authenticators will inherit their attributes and methods from an abstract class (a class meant to be inherited) for overriding. This abstract class is described in the *Code components* chapter.

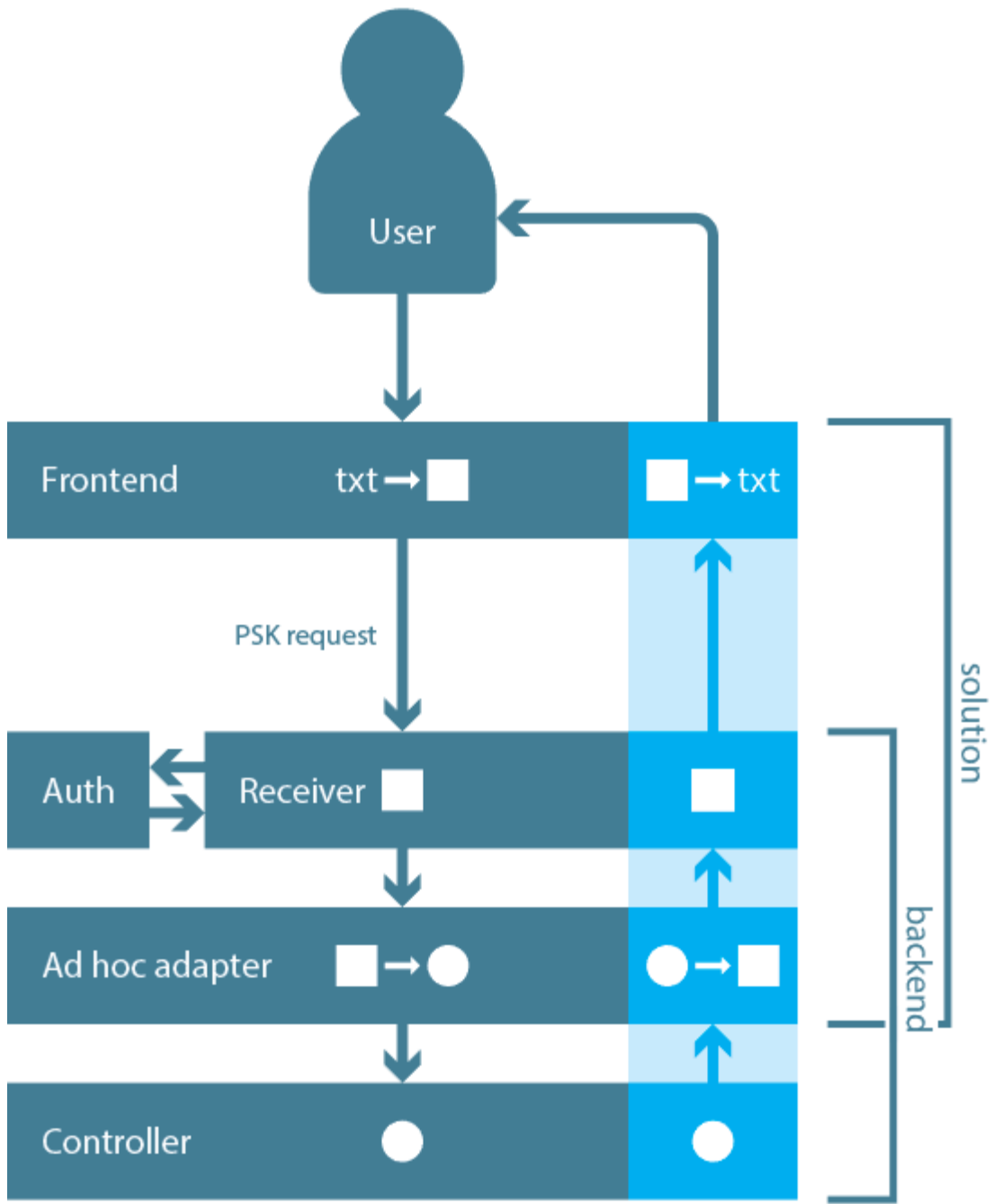
Ad hoc adapter

Ad hoc means “for this” in Latin. Because we want the solution to be adaptable and not reliant on a specific manufacturer of access points, for example, we need another replaceable module. The ad hoc adapter will be a code component that tailors a PSK request for the a specific network management system, such as HiveManager. This is needed because each such system may use different methods for generating PSKs. One may use an API endpoint which can be easily called, while another (such as hostapd) may use a configuration file and needs to be restarted for the changes take effect. This custom logic is

performed in the ad hoc adapter, which will inherit an abstract adapter class that tells the ad hoc adapter which methods must be implemented. This class is described in the *Code components* chapter.

Illustration

The following illustration shows the cooperation between the various modules. The symbol in the middle of a component signifies the type or format of data it accepts coming in. The symbol in the cyan area to the right signifies the type or format of data it accepts coming back from the next module in the chain and/or what type or format of data it will return to the module before it in the chain. An arrow signifies a conversion from one type or format to another.



In this example, the controller expects PSK requests with a “circle” format. Therefore, the ad hoc adapter converts the native “square” format to a “circle” format. The circle format may also symbolize that the controller expects a certain routine to be performed, such as the changing of a configuration file. In that case, the ad hoc adapter interprets the square data and performs the necessary circle operations on the configuration file. Once the PSK has been generated, the new PSK may or may not be returned to the ad hoc adapter. The circle format may then represent the need for the ad hoc adapter to investigate a log file on the controller device, for example, or extract a key in the returned JSON format. This uncertainty is what the ad hoc adapter deals with. The extracted PSK is then returned to the receiver in the native square format. The frontend finally converts the response from the receiver to a string that is displayed on the user interface, if this is the desired behavior.

Note that “circle” and “square” have no real meaning, but are meant to illustrate the fact that not all controllers will return the newly generated PSK in the way the receiver expects.

OAuth 2.0 Authentication

Our solution will require that the user authenticates using Feide, which merged with Dataporten in October, 2018. We may use the names Dataporten and Feide interchangeably. Feide is an Identity Provider (and we are the Service Provider) that supports OAuth 2.0, a widely used protocol for authorization.

Service Providers with Feide integration require that the user authenticates using Feide’s SSO page (unless other means of authentication exist). This means that we cannot accept Feide credentials in the *authentication_data* field of the request and simply query Feide’s servers. As a Service Provider, we are not allowed to be in possession of the user’s secret password. Our authenticator must verify that the user is authenticated without receiving the credentials. This can be achieved using sessions, which are supported in Django by default. We will use a database for storing sessions.

This means that our Feide-specific solution will require additional views that handle, among other things, the verification that the client supports and accepts the storing of cookies, and the receiving and storing of user attributes from Feide’s systems.

A more detailed description of how OAuth 2.0 is used by Feide can be found in their documentation at https://docs.feide.no/developer_oauth/ (retrieved on March 20th, 2019).

User interfaces

The backend requires several pieces of information from the user. Some of these pieces of information, however, do not require an input field, as they will be provided by Feide once the user is authenticated. What we require that the user specifies is their preferred email address, a descriptive name for the device to be connected, and a choice regarding whether the newly generated PSK should be delivered by email or not.

As for choice of graphical design and color, we have opted for a minimalistic, two-color page for simplicity and due to personal preferences. The user interface may be customized, so long as the required fields (as expected by the backend) are present.

Here, the user’s first name is Ola, which is one piece of information given to us by Feide after the user has signed in using their SSO.

Velkommen, Ola

E-postadresse

Kort beskrivelse av enheten

Send passordet via e-post

Hent passord

Form validation

While the backend will contain methods for data validation, it is generally a good idea to perform validation on the frontend as well. Frontend validation cannot be trusted, as the user can bypass it easily by making changes to the markup and scripts that are loaded into memory on the client's side, but it gives the user immediate feedback on the fields without having to wait for a request to fail. It also prevents needless traffic from reaching the server.

The consistency of validation on the frontend with the backend's validation must be ensured for each implementation of the solution. If we were to expand the scope of the project, we could devise a solution where the programmer defines validation criteria on the backend, which are then fetched by the frontend and injected into HTML form elements. This would be an endeavor for the future and will not be featured in our solution.

Backend code components

In this subchapter, we describe designs for the various components and modules of the system with the highest degree of detail possible at this stage.

Basic explanation of Django REST framework

Without going into too much detail, it is helpful to have an idea of how Django REST framework works. This is a Python framework that is perfect for backends. It allows us to specify any number of URLs, and each URL points to a certain *view*. A view is either a class or a function that handles HTTP requests. For example, a *GET* request to the URL `hostname/users/` may return a list of users in the system, and a *POST* request to the same URL may create a new user. The logic is performed by the URL's view. The framework handles much more than this, and includes concepts such as authentication backends, user

management, permission requirements and more, but for our purposes, most of this functionality will be optional and may be implemented if required in a given implementation.

Introduction to types and concepts in RESTful APIs and Django REST framework

In order to clarify the description of components and architecture, we will briefly describe some types and concepts that are central to RESTful APIs and the Django REST framework.

RESTful API

REST stands for Representational State Transfer. A RESTful API is an API (application programming interface) that lets a user or programmer interact with the service using HTTP requests such as GET, POST, or DELETE. APIs are commonly used by programs, such as a web page or an app (the frontend) to get, create or edit objects that exist in the backend, which can only be reached and interacted with through precisely defined and strictly controlled “openings” called endpoints.

Endpoints

An API *endpoint* is the end of the communication channel that faces the user or programmer. In other words, a user or programmer — usually by means of interacting with a frontend application — may send an HTTP request to an endpoint and receive a response. A specific API endpoint is typically reached through a specific URL or collection of similar URLs.

View

In Django REST framework, a view is a class or function that handles HTTP requests, usually for a specific, single endpoint. A response is returned to the client after handling.

Request

The Request class is a Django REST framework class that represents an HTTP request. Because of Django’s native handling of requests, there is no need to convert the raw request data to a Request instance ourselves. We may safely expect the Request object to be sent as an argument to our view (if these views are subclasses of the built-in `APIView` class).

Response

Once a view has finished processing a request, it is typically expected to return a Response object. A Response may include custom data, as well as an HTTP status code. HTTP status codes include 200 OK, 404 NOT FOUND, 400 BAD REQUEST, and many more². The Response object is converted from Django REST framework’s representation to a HTTP response and sent to the client that made the original request. This concludes the life cycle of the request (though it may pass through additional *middleware*, a concept we will not explain here).

² An up-to-date list of HTTP status codes can be found at: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes (retrieved on March 6, 2019).

Description of components and architecture

Our general solution requires three abstract (meaning they are meant to be inherited and cannot be used *as is*) classes: An *Authenticator* class, an *AdHocAdapter* class, and an *IoTConnectView* class.

Authenticator

This class will contain the methods that determine if the user is allowed to proceed. This can be determined by the credentials passed (if any) to it, or by verifying a token, for example. In our case, the class that extends this abstract class will verify that the user is authenticated with Feide. The class will contain two functions: 1) `is_authorized`, and 2) `authenticate`.

Both functions are expected to return a Boolean (true or false) value, where *false* indicates that authentication failed, and *true* indicates that it passed. The former, `is_authorized`, is meant to be overridden (implemented by the child class). If it is not overridden by the child class, an exception will be raised. The overridden function should perform the check itself and return *true* if it passes, or *false* if it fails.

The latter function, `authenticate`, is not meant to be overridden. It calls the `get_validated_data` as described below (in `ValidatorMixin`), sends the validated data to the `is_authorized` function, and returns the resulting Boolean value.

AdHocAdapter

This class handles the custom logic required to generate the PSK. It, too, will contain two primary functions: 1) `perform_generation`, and 2) `generate_psk`.

The former, `perform_generation`, is meant to be overridden. If it is not overridden, an exception will be raised. The function is expected to return a `Response` object. In other words, the logic in this function, when overridden, determines the response that is returned to the client after the request.

The latter, `generate_psk`, is not meant to be overridden. The function first calls the `get_validated_data` function (again, as described below, in `ValidatorMixin`), then calls the `perform_generation` function, passing the validated data as an argument. The result of `perform_generation` (a `Response` object) is returned.

IoTConnectView

This is the class that will handle requests for new PSKs. The class will inherit Django REST framework's `APIView` class, which implements numerous handy functions and attributes, so that we need not.

The primary functions of the `IoTConnectView` are: 1) `dispatch`, 2) `post`, and 3) `process`. In addition, it will declare two mandatory (meaning their value cannot be `None` — known as *null* or *nothing* in other programming languages) class attributes: 1) `ad_hoc_adapter`, and 2) `authenticator`. As the names imply, the `ad_hoc_adapter` attribute is expected to be an instance of a class (or the class itself, depending on decisions made in the execution stage) that inherits the abstract

AdHocAdapter class. Likewise, the `authenticator` attribute is expected to be an instance of a class or the class itself that inherits the abstract Authenticator class.

The `dispatch` function is a function that is implemented by `APIView`. In other words, `lotConnectView` will override `APIView`'s `dispatch` function. The `dispatch` function is called when a request is received, and its purpose is to find the appropriate function of the view to call (the function that handles requests of this HTTP method). `lotConnectView`'s `dispatch` function will first validate the attributes of the class according to the following rules: 1) `ad_hoc_adapter` cannot be `None`, 2) `authenticator` cannot be `None`, and 3) the instance that is the value of the `authenticator` attribute must have a method called *authenticate*. We will not require that our abstract Authenticator class is used by the authenticator, but we *do* require that whatever class is used for the authenticator instance has a function named *authenticate*. Furthermore, the `dispatch` method will validate the HTTP request, ensuring that the required `authentication_data` and `generation_options` are present. If this validation fails, an exception will be raised. Otherwise, the request will be forwarded to the method handler. This method is not meant to be overridden further.

We will let the programmer decide which HTTP method should be used for the generation of PSKs, but it will default to `POST`, and we will assume that `POST` is used in this description. If the HTTP method of the request is `POST`, the `dispatch` method of `APIView` will find a function in the class named *post* and use it as the handler for the request. If the programmer or user attempts to send a `GET` request, for example, a 405 Method Not Allowed response will be returned. If the request method is `POST`, the `post` method will be called. The `post` method will call the `authenticate` method of the authenticator, and the `generate_psk` method of the ad hoc generator. It will raise an exception upon failure, or pass the response generated by `generate_psk` to the `process` function upon success. The `post` function is not meant to be overridden.

The `process` function may optionally be overridden by the programmer. This function is expected to return a response, and will return the response passed to it (as created by `generate_psk`) by default. This is the last step in the request's life cycle, where the programmer is allowed to create any database entries required of their implementation, or whatever else is required.

The `AdHocAdapter` class and the `Authenticator` class both have some functionality in common. Luckily, Python classes can inherit multiple base classes. Classes with a small set of members that may be required of multiple other classes are typically called *mixins*. We will use one such mixin, which we will call *ValidatorMixin*. This way, we can adhere to the *don't repeat yourself* (DRY) principle.

ValidatorMixin

This mixin will have two methods: 1) `validate_data`, and 2) `get_validated_data`.

The former, `validate_data`, is meant to be overridden. If it is not overridden by the child class, an exception will be raised. This function will receive the authentication data or the generation options (which are sent in the request data) and validated. Validation may include ensuring that all fields required of the specific implementation (such as a Feide+HiveManager implementation) are present in this data. If validation fails, an exception should be raised. The function should return the validated data, which may differ from the input data.

The latter, `get_validated_data`, is not meant to be overridden. This function will call `validate_data`, and then perform whatever mandatory validation is necessary, such as checking that the validated is actually returned.

Because our solution is specific to Feide and Aerohive's HiveManager/PPSK, we must design a few more classes. We will call these classes *HiveManagerAdapter*, *FeideAuthenticator*, *ConnectView*, and *DataportenRedirectView*. *HiveManagerAdapter* will inherit *AdHocAdapter*, *FeideAuthenticator* will inherit *Authenticator*, and *ConnectView* will inherit *lotConnectView*. *DataportenRedirectView* will inherit the built in *APIView* class. These names have been chosen somewhat arbitrarily, but they adhere to the conventions we are used to. The logic required to customize the solution to a Feide+HiveManager solution will be contained solely within these classes.

HiveManagerAdapter

As required by the *AdHocAdapter* class and *ValidatorMixin*, this class must implement two methods: 1) `validate_data`, and 2) `perform_generation`. The former will ensure that the data required by HiveManager's API is present or calculable, and the latter will call HiveManager's API endpoint for generating PSKs, and return an appropriate response.

FeideAuthenticator

As required by the *Authenticator* class and *ValidatorMixin*, this class must implement two methods: 1) `validate_data`, and 2) `is_authorized`. The former will ensure that the data required by Feide's API is present or calculable, and the latter will call Feide's API endpoint for validating Feide credentials, and return a Boolean value indicating whether or not the authentication was successful.

ConnectView

As required by the *lotConnectView*, this class will simply set the value of the `ad_hoc_adapter` to an instance of *HiveManagerAdapter*, and the value of the `authenticator` attribute to an instance of *FeideAuthenticator*.

DataportenRedirectView

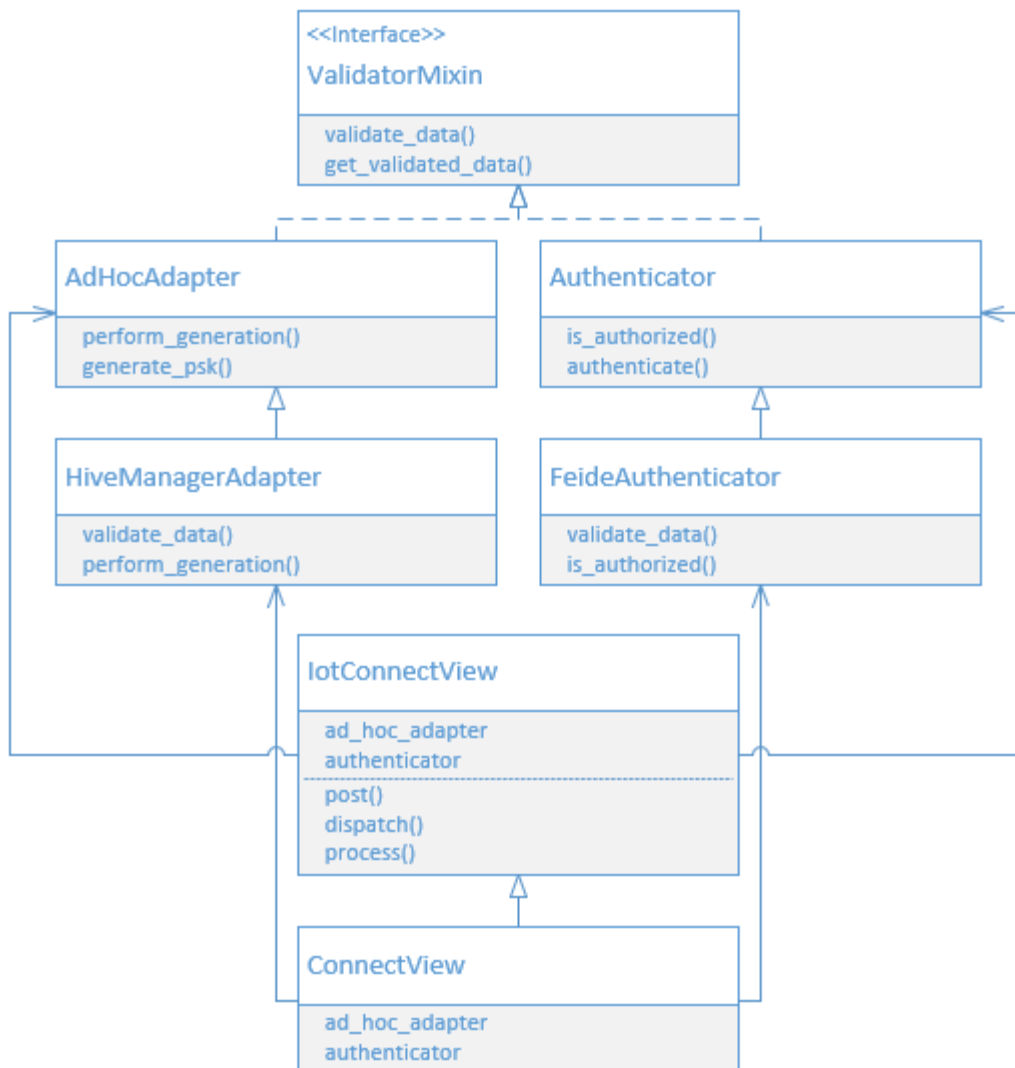
Because OAuth 2.0 requires that we provide a redirect URI to which the authenticating user is redirected after authentication has succeeded, we will add another view that finalizes the authorization. Here, a code supplied by Feide will be used to retrieve an access token that we can use to get data about the user, such as their name. After retrieving the access token, the view will create a session for the user and store the access token there. After the user has been redirected to the frontend, where they have to fill out a short form before requesting a PSK, the

authenticator will be able to find this session and validate the access token before generating the PSK.

Besides defining the URLs of these views and customizing the user interface, this is all that is required of a custom solution.

Class diagram

The relationship between these classes is illustrated by this UML class diagram. Dashed lines signify the implementation of the ValidatorMixin mixin (often called interfaces in programming languages where a class may only inherit from a single base class). Solid lines with closed, white arrowheads signify inheritance, and solid lines with open arrowheads signify direct associations (in this case meaning that there are references to instances of the associated type within instances of the class that the arrow is coming from). We have not included method parameters and their expected types at this time. If a dashed line exists in a class, the members above it are attributes and members below it are methods.



Reasoning

This relatively simple architecture eliminates the need for repetition of code and leaves the programmer with only three classes needing to be overridden for a complete, customized solution. Each of these three components have different purposes, and therefore should be contained within separate components. It utilizes the tried and true architecture of Django REST framework as well, with the view class acting as the receiver of a request.

Django REST framework's *APIView* class features the built-in *authentication_classes* and *permission_classes* fields. We have decided not to make the authenticator an authentication or permission class as we will not be using the built-in user management functionality. If the programmer wishes to implement this solution into a system that features user management (including a *users* table in the database), they may optionally use these fields in the same way as they are used elsewhere in their system. Our system has no concept of a *user*, and all users will either be anonymous users or superusers.

We believe this architecture will be relatively easy for us to develop and simple for the programmer to implement while providing a great degree of flexibility.

Data templates

The receiver will expect the data sent from the frontend to be a dictionary containing two keys: 1) 'authentication_data', and 2) 'generation_options'. The receiver does not care about the structure of the data of these keys, but forwards the value of the 'authentication_data' key to the authentication module for validation, and the value of the 'generation_options' key to the adapter for validation. If validation fails, a 400 Bad Request response is returned to the client. If authentication fails, a 403 Forbidden response is returned.

Deserializers are classes that are used to create objects. A user deserializer, for example, could feature fields such as *email*, *username*, *first_name* and *last_name*, and each of these fields may be marked as being required or not. By passing the data received to a deserializer and checking if the deserializer considers the data valid, we will have achieved a more elegant form of validation than the alternative, which is to implement a series of statements that verify the presence of specific fields and values in the data. We will consider implementing validation through deserializers if there is time.

Frontend details and code components

The frontend will feature the design as shown under the *User interfaces* subchapter. When the user is redirected here after authentication with Feide, their name will be sent along as a query string. In other words, the name of the user will be visible in the URL as such: `hostname/index.html?name=Ola`. This value can be retrieved using JavaScript and used in the welcome message that is displayed above the form.

When the user has filled in the fields of the form and clicked the button to request a PSK, the data in the form's input fields will be structured according to the backend's expected format thusly:

```
data = {
  'authentication_data': {...},
  'generation_options': {
    'email': [user's email address],
    'device_type': [descriptive name of the device],
    'deliver_by_email': [true or false]
  }
}
```

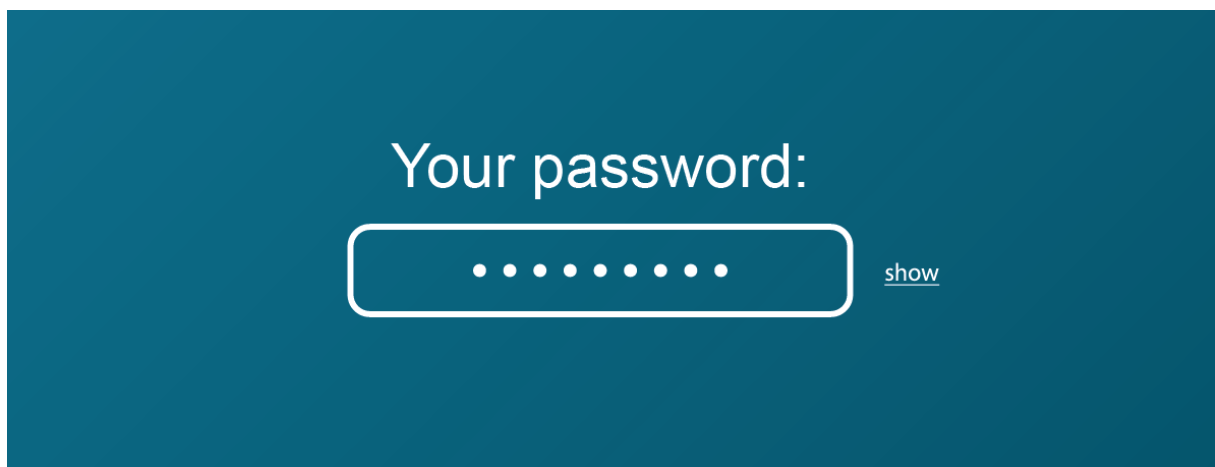
We do not plan for `authentication_data` to contain any data, as the user's session will contain their verifiable access token.

The request will be sent using Ajax (short for asynchronous JavaScript and XML) asynchronously, and the generated PSK or any error messages will be retrieved without a redirect or refreshing the page. The returned data will then be shown appropriately on the page, and this, too, will be done using JavaScript.

In summary, there are three code components of the frontend: 1) A script to populate the page with data from query strings upon arrival, 2) a script to send the POST request asynchronously, and 3) a script or function that updates the page with the response from the POST request.

Success and error handling

Upon the successful generation of a new PSK, the new PSK will be displayed thusly:



If the generation was not successful, the result will depend on which exception occurred. We have identified several possible reasons for failure, and we have placed these in one of two categories: 1) Exceptions caused by user error, and 2) exceptions caused by internal errors.

The latter category is not expected and would be caused by improper or inconsistent validation, or an issue with the multi-PSK provider or Identity Provider.

These are the identified exceptions caused by user error:

Unauthorized

The user may click the submit button after prolonged absence from the computer has rendered their authentication invalidated. The standard response to such exceptions is the 401 Unauthorized status code, and the response will usually include instructions on how to authenticate in its headers. Because we are using an asynchronous call, we must add the logic required to redirect the user to Feide's SSO page. After the user has re-authenticated, they will be redirected back to the frontend, where they may try again.

Forbidden

The 403 Forbidden status code is usually returned when a user attempts to access a resource or perform an action that they are not authorized to access or perform. In this prototype, a user will be authorized as long as they have a Feide account, and being unable to authenticate using Feide will be handled by Feide, and not our application.

Still, HiveManager somewhat confusingly returns a 403 Forbidden status code when attempting to generate a new PSK and passing an already used username. HiveManager also has a somewhat confusing and undocumented way of determining which field in its user model constitutes the unique identifier. For example, if an email address is passed but no username, the email address will be the identifier. If a full name is present, but no email address or username, the full name will be used as the identifier. If the username is present, it will be the identifier.

Because we wish to allow one user to create any number of PSKs at this time, we have found it necessary to append the date and time to their username to avoid such conflicts. Still, we may limit the number of PSKs a user is allowed to create in a given timespan. This could be done simply by rounding the date and time to the closest 10 minutes, for example. In that case, user identifier conflicts could easily happen if the user generated new PSKs too frequently. If we decide to limit the number of PSKs a user may generate in this way, we will display a message to the user that they must wait before creating a new PSK.

These are the possible exceptions caused by an unexpected internal error we could identify:

ValidationError

The ValidationError class is an exception class in Django that, if raised during the life cycle of a request, will return a 400 Bad Request response by default. If the classes in our solution are set up correctly, this would only occur if required fields are missing from the request. Required fields would only be missing from a request if the frontend's validation was inconsistent or lacking. The user should be alerted of missing information before the request is sent.

If we decide to use deserializers as validators, the deserializer class we choose might give us some information about which fields are missing or not within parameters. We will look

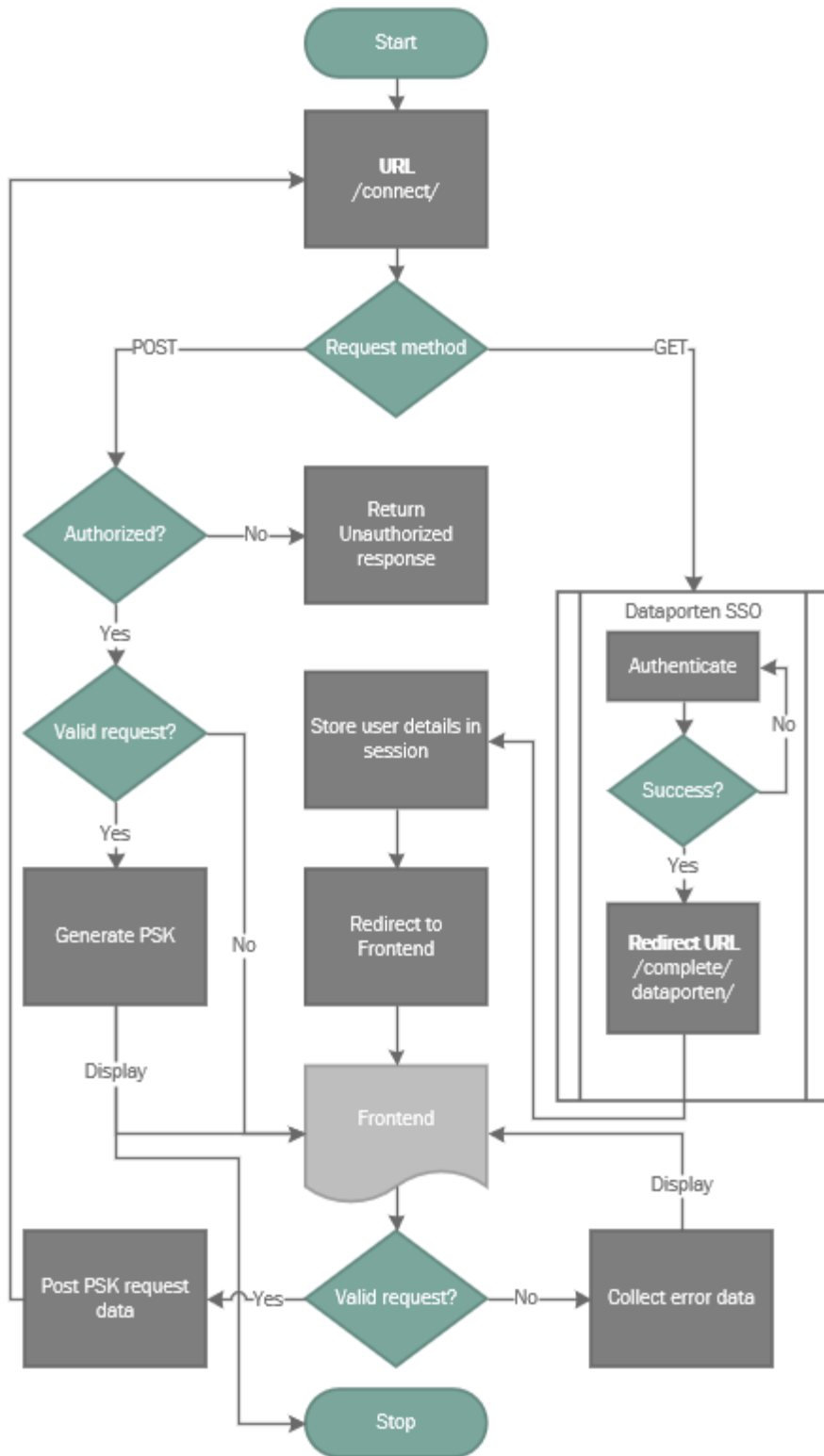
into the possibility of letting the user know exactly which fields failed the validation but, if this is not possible, we will simply ask the user to double-check their input.

Unknown error

Any exception raised during a request that is not handled by Django's default error handler will result in a 500 Internal Server Error. This is the final catch-all for unhandled exceptions. We cannot allow the user to know what went wrong (by enabling debugging), as this poses a security risk. Instead, we will let the user know that something went wrong and that they should alert an administrator.

Flow

This flowchart shows the intended steps in the process of generating a PSK:



The user first attempts to connect to the /connect/ URL. This view expects the request method to be POST, but a user visiting a URL using their browser results in a GET request.

The view will know to redirect the user to Dataporten for authentication if the request method is GET.

If authentication on Dataporten's SSO page is unsuccessful, the SSO page will alert the user, and no redirect will occur. Otherwise, the user will be redirected to a URL on our backend that ends with `/complete/dataporten/`, as specified in Dataporten's documentation. This redirect will come with some information about the user, such as their name.

The view that handles this redirect will store the user's data in a session before redirecting the user to the frontend, where the form is presented for the user to populate with further required information. The frontend performs its own validation and will inform the user if any fields are invalid. If the data entered by the user into the form's fields is valid, the frontend sends a POST request to the backend's `/connect/` view.

The backend verifies that the user is indeed authenticated. If the user is not authenticated, they are not authorized, and a 401 Unauthorized response is returned. If they are authenticated, they are authorized, and the request is validated on the backend for good measure. If validation fails, an appropriate error message is returned to the requesting frontend and displayed on it. If validation passes, the PSK is generated and returned to the frontend, where it is displayed. This marks the end of the process.

Database

Databases are used extensively by most Django projects. Even if we don't plan on having any models (classes with corresponding database tables, such as users, articles or species), we need a database for various reasons. These reasons include the possibility of third-party libraries requiring database tables, migrations (incremental changes to the structure of database tables), and sessions. We will be using sessions in our custom solution.

The two viable options are PostgreSQL and SQLite. SQLite does not support concurrency. We will be using PostgreSQL for our database.

Security measures

Because sensitive information is communicated in this system, security is the top priority. In fact, security should have a higher priority than functionality and indeed having a working solution. In addition to following standard safety precautions, such as disabling debug mode on the Production branch, we will implement the following security measures.

Expiration of old pre-shared keys

The ability to expire old pre-shared keys depends on the controller component of the system. If the controller software allows the administrator to specify a duration, this duration should be set by the ad hoc adapter module and may optionally depend on custom data sent from the frontend. If the controller software does not support setting a duration upon the creation of the new PSK, then a periodic task must run that deletes keys with a certain age. The latter

solution may require that PSKs are stored in a database, which lowers security because anyone with access to the database will be able to decrypt communication between devices using the PSK and the access point. The question of whether or not this feature should be implemented is a complex one whose answer depends on the circumstances. The storing of PSKs in a log or database should optionally be handled by the ad hoc adapter, and a periodic task that calls a `expire_keys()` may be created. The method `expire_keys()` should, if implemented, be overridable, and any logic required to delete old PSKs should be placed there. We will make further decisions regarding this in the execution stage.

Generation of strong pre-shared keys

It is crucial that dynamically generated pre-shared keys are not predictable. Random value generators typically use the exact time and date (including highly insignificant units of measurement such as microseconds) as their seed (an input value that results in a certain output value) unless a seed is specified. Version 3.6 and newer of the Python standard library include the `secrets` module. The function `secrets.token_urlsafe()` generates random strings that are considered sufficiently unpredictable. Because the exact time is presumably at least part of the seed for this generator, any information stored about the creation of the PSK should not include any unit of measurement of time less significant than the second. Additionally, a `wait` function with a pseudo-random duration may optionally be called after each PSK request in order to reduce the susceptibility to brute force attacks, and to make sure that packet sniffers cannot easily predict the exact time that the PSK generator is called by looking at the timestamp of PSK request packets.

The absence of units of measurement of time less significant than seconds must be ensured by the programmer developing the ad hoc adapter. We cannot easily enforce such a policy without drastically expanding the scope of the project.

Furthermore, HiveManager does not require that we specify a PSK when generating one. A random PSK is generated by default, and we are not sure if HiveManager allows us to generate a PSK ourselves in the first place. We will rely on HiveManager's random value generator. In other words, the randomness of generated passwords must also be ensured by the programmer. A short pause will be implemented in the abstract ad hoc adapter class. We may decide to make this pause optional.

Obscurity

Security through obscurity is the reliance on hiding sensitive features and information as a means of achieving security. This is generally considered bad practice, for good reasons. By designing a secure solution *before* obscuring these theoretically secured features, however, we are — to some degree — securing our system against vulnerabilities that we are unaware of and would not think to secure in the first place. It is not uncommon to hide the URL of admin pages, for example.

The Django admin site is commonly found at `host/admin/`, but this may be changed for any project by the programmer. An attacker who wishes to break into the system will easily find the admin site. We will place the admin site elsewhere. The URL will act as a password of

sorts, and it should only be known by those who will be managing this system. Changing the location of the admin dashboard does not give any increased degree of security if its location is known to the public, which it will be if our public Git (<https://git-scm.com/>) repository reveals the location. Therefore, this measure should be implemented by the individual programmer if it is considered desirable.

SSL

Because authentication credentials and secret keys will be transmitted over the internet, it is crucial that communications between the client and the backend be encrypted. We will enforce HTTPS by redirecting HTTP to HTTPS. This is achieved by changing several settings in a Django project and must be done as part of the implementation process by the programmer. We will include these steps in an installation guide that we will be attached to the execution report.

Hiding of secrets from Git repositories

Django projects have secrets in their settings, and our solution will contain many more secrets than a standard, newly created Django project. These settings exist in python files, and there is no separation between secret settings and non-secret settings. Because we need most non-secret settings to be consistent across all branches, we will implement a settings structure (using Git's list of ignored files) that confines secret settings in local, Git-ignored files that inherit other settings from a non-secret *base* settings file. These secret files must be kept consistent across branches manually if necessary.

Secrets cannot be pushed to a public Git repository, nor to a private repository that is made public, even if they are removed at a later date. This is because the addition and removal of the secret will be recorded in the commit history. It is possible to purge GitHub's servers of all mentions of the secret, but once a secret has reached a public Git repository, it must be considered compromised. This settings hierarchy prevents us from ever having to publish secrets to our Git repository, and the base settings file can be published safely.

Version control with Git

Git is a system for version control. We will use Git for all our code and markup. Our backend's Git repository will have a Development branch (the *master* branch), a Testing branch, and a Production branch. Commits are first pushed to a *feature* branch, which is a branch created specifically for the development of a feature, a bug fix or similar. Once work is completed on the feature, a pull request is created, requesting that the feature branch be merged into the Development branch. Before this can be done, reviews and tests must pass. When the Development branch is in a satisfactory state, we may merge the Development branch into the Testing branch. After we have performed a satisfactory amount of testing against the Testing branch, we may merge the Testing branch into the Production branch. This flow helps us ensure that the Production branch is always in a working state.

Testing

Before the Development branch can be merged into the Testing branch, all unit tests must pass. Unit tests are pieces of code that perform some action in the system and make assertions regarding the result of the action. If the actual result differs from the expected result, the unit test does not pass. We will configure our project to automatically run unit tests using CircleCI upon every new commit to the Development branch. If any of the tests fail, the commit cannot be merged into the Testing branch.

Summary

Our solution will consist of a frontend and a backend. These will exist on separate virtual servers.

The backend will be built using Django REST framework, and will consist of several main components: The receiver (a view), a redirect view (the landing point after Feide authentication, redirecting the user to the frontend), an authentication (responsible for checking that the user is authorized to generate PSKs), and an ad hoc adapter (responsible for the generation of the PSK itself). The frontend will consist of simple, CSS-styled HTML and JavaScript (responsible for making asynchronous requests).

In this document, we have outlined plans for testing, version control, increased security, data formats, class hierarchies, the flow of information, hardware and software specifications, and more. These plans are a crucial step in the project and will prepare us when the time comes finally to develop the solution.

Deviations from the prestudy report

In the prestudy report, we declared a goal of having finalized the design report by the end of week 11. It is not the beginning of week 13. This means that we are behind schedule, and could not satisfy the process goal of experiencing no deviations from the progress plan exceeding 3 days.

Furthermore, the prestudy report declared that a web dashboard would be necessary for authentication. While a web dashboard is necessary, it is not necessary for authentication purposes, as Dataporten's SSO serves this purpose. It would have been necessary if we proceeded with the "hotspot" solution. We have amended the prestudy report.

Conclusion

We are now closer to having a secure, scalable and simple solution for the problem at hand: The secure onboarding of IoT devices that only support WPA2-Personal. We are very excited to have been enabled to take this design approach, thanks to the involvement of Aerohive.

In this document, we have planned our solution in as much detail as is realistically possible and viable. There is undoubtedly an infinite number of valid approaches, and we believe that our design pattern is one of them. With a highly customizable solution, we believe that this solution will satisfy a broad range of requirements.

This document marks the end of the design stage. We will now be moving on to the execution stage, in which the goal is to implement these designs in a fully working solution.

Amendments

Date	Details
May 7, 2019	The arrows signifying extensions and interface realizations in the UML diagram showing the planned architecture of the backend were pointing in the wrong direction. They were flipped.

Bibliography

Home - Django REST framework. (n.d.-b). Retrieved March 25, 2019, from <https://www.django-rest-framework.org/>

Aerohive Networks. (n.d.). DATA SHEET: AP122 [AP122 specifications (PDF)]. Retrieved March 25, 2019, from https://www.aerohive.com/wp-content/uploads/Aerohive_Datasheet_AP122.pdf

Aerohive Networks. (n.d.-b). Aerohive Networks Inc API [API documentation (requires account)]. Retrieved March 25, 2019, from <https://developer.aerohive.com/docs/api-documentation>

UNINETT / Feide. (n.d.). Feide documentation [Feide documentation]. Retrieved March 25, 2019, from <https://docs.feide.no/>

IETF OAuth Working Group. (n.d.). OAuth 2.0 — OAuth [Documentation]. Retrieved March 25, 2019, from <https://oauth.net/2/>

AUTHENTICATION IN THE INTERNET OF THINGS

Execution report

Part of a Bachelor's thesis

**Presented to the Institute of computer technology and informatics
of the Norwegian University of Science and Technology
by Magnus Bakke & Liang Zhu**

Submitted in partial fulfillment for Bachelor's degree of
Informatics with specialization in network administration
during the year 2016–2019

Introduction

This project began when we, the candidates, approached Uninett AS — the state-owned company responsible for IT services for research and education in Norway — to inquire about the subject of authentication in the internet of things (IoT). We hoped to be able to write our Bachelor's thesis on this topic, and both Uninett and our supervisor (Stein Meisingseth) at the Norwegian University of Science and Technology (NTNU) accepted the proposal.

The task given to us by Uninett specifically asked us to research possible solutions for secure authentication of IoT devices that do not support the secure WPA2-Enterprise standard. Furthermore, we were asked to select a possible solution and put it to the test.

The search for solutions began in the initial research stage, which yielded two main solutions and a few specific variations of these. The clearly superior solution, as outlined in the research report, involves technology commonly referred to as multi-PSK, Private PSK, or other variations thereof. If we were to select this solution for execution and testing, we would have to borrow access points that support this technology, or settle for the much less practical solution offered by hostapd (which requires restarting the service every time a PSK is created). Aerohive Networks is among the few providers of this technology, and they kindly sent us two access points for testing. Having the necessary hardware, we could begin designing a solution in the design stage.

The design stage produced the design report, which gave detailed plans for most aspects of the solution. Upon its completion, we would enter the most exciting stage of the project: The execution stage.

The execution stage began with enthusiasm and eagerness. We promptly began developing prototypes for both the backend and frontend. We soon found that we were abandoning our own requirement of using Scrum, which in turn made it difficult to measure progress and estimate the remaining workload. We soon adopted the methodology we had used in the design stage.

In general, we prioritized the more pressing and substantial issues, such as the development of the *receiver* component (as described in the design report) of the backend and the form and request sending functionality of the frontend. These issues were mostly completed with great efficiency and speed.

Issues started to arise when it was time to move our code and static content to servers for the testing and production environments. We spent nearly a week trying to get a working Apache installation. None of the numerous guides we sifted through would be applicable to us on account of the diversity of Linux environments. When we finally embraced the newer solution named Caddy, we got everything up and running within a couple of days.

Another issue involved difficulties with Cross-Origin Resource Sharing (CORS) in conjunction with asynchronous requests sent with JavaScript from some browsers. We quickly found a workaround for this.

The majority of our available hours in the execution stage have been spent working on this report, which — alongside the other reports — is part of what is undeniably the most important aspect of the product. While the solution we have prototyped serves as a proof of concept, the reports summarize our research and findings on the viability, advantages and disadvantages for different possible solutions.

Summary of the problem

A personal wireless network is typically secured using WPA2-Personal. This involves setting a password that is required by any user of the network for access. This password is called the pre-shared key (PSK), as it is shared with those whom are trusted by the network's owner. In other words, a guest in someone else's home needs the host to give them knowledge of the PSK before they are able to connect to the wireless network.

The PSK is used as the encryption key. This means that anyone who does not have the PSK will be unable to decrypt the communications between the access point and the user. It also means that anyone who *does* possess the PSK will be able to monitor other users on the network. This is unacceptable in enterprise settings with many users. A PSK cannot be considered secret and secure when shared among dozens or hundreds of employees, and not every employee can be trusted to not monitor other users' communications.

The solution is the WPA2-Enterprise standard, which uses public-key cryptography. Public-key cryptography involves a pair of keys: The public key and the private key. The public key can be used to encrypt messages (such as dynamically generated PSKs) that only those who possess the corresponding private key may decrypt. This concept can also be reversed, so that the public key is used to decrypt a message that was encrypted using a secret, private key. In this way, users can create signatures that cannot be forged. Thus, there is no need for further authentication. This is the scheme used by most enterprises.

Problems arise when manufacturers of IoT devices neglect the enterprise world and market their products for personal use. Many IoT devices do not support WPA2-Enterprise, and so require a "master password" to be inputted.

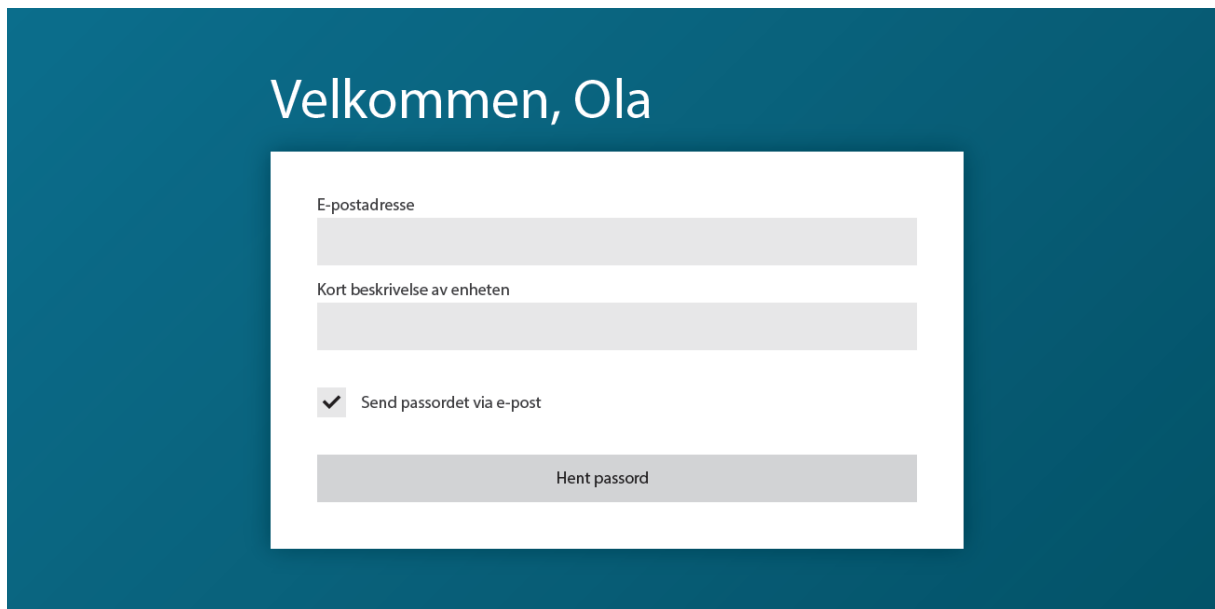
The best existing solution to the problem of lacking WPA2-Enterprise support in IoT devices involves dynamically creating new PSKs upon request. A user may bring an IoT device to their institution's offices, request a unique PSK for their device, and use that PSK as the master password. With PPSK, as Aerohive has named this technology, we can create any number of PSKs dynamically, and the communication between the access point and the device is encrypted using the unique PSK. This does not require that each device communicates on a different SSID.

While we now have access to this technology, we must still develop a web application that handles requests for new PSKs, forwards them to the HiveManager API, and returns the PSK to be displayed on a user interface. The complete solution, including the frontend, backend, security measures and much more, was designed in the design stage.

Summary of designs

The designed solution consists of a frontend (the pages that the user interacts with using their browser), a backend (code running on a server that performs logic “behind the scenes”), the environment, which includes third-party services like Dataporten/Feide and HiveManager, the ways in which these components will communicate, and more.

The frontend is best summarized using a picture of the design:



Velkommen, Ola

E-postadresse

Kort beskrivelse av enheten

Send passordet via e-post

Hent passord

Please note that Uninett has decided that the email address associated with the user’s Feide account should be used instead. Therefore, there will be no input field for the user’s preferred email address.

This page can be accessed at a memorable URL in the user’s browser. After filling out the form, the user clicks the button. This leads to the frontend sending a request with the form data to the backend, which itself constructs a request. This request is sent to HiveManager’s API, which returns a newly generated PSK. This PSK is returned to the frontend, which displays it thusly (though in Norwegian):

Your password:

[show](#)

The backend is designed to be versatile and customizable. This means that it will not make assumptions about its environment, such as which identity provider (in our case, Dataporten/Feide) or PPSK provider (in our case, HiveManager) is used. This will be achieved with the designed class hierarchy and design pattern.

Scope and purpose of the document

This document will give an account of how the designs from the design stage were executed and realized as a working solution. Unimportant details will not be included. Unimportant details are those that are not required of the designed solution, such as the purchasing of domain names, the creation of virtual machines, or the backing up of data. The execution of stylistic choices through HTML and CSS will also not be described in detail.

Finally, the document will describe how the product was tested and the results of these tests. We will also propose further developments that are beyond the scope of this project, as well as discuss the viability, future, and longevity of the technology.

After reading the document, the reader should understand how such a solution can be developed and deployed. The document serves as a guide as well as an account of the activities undertaken in the execution stage. It should also provide outlooks for the benefits, shortcomings and future of the technology.

Contents

Introduction	2
Summary of the problem	3
Summary of designs	4
Scope and purpose of the document	5
Contents	6
Definitions	8
Setup of Django	9
Installation	9
Code editor	10
Project structure	10
Apps and URLs	10
Other packages	13
Setup of Git	14
Branches	14
Branch protection rules	15
Hiding secrets and localizing settings	16
CodeFactor	17
Setup of environment	20
Servers	20
Installation and configuration of Caddy and gunicorn	21
Databases	24
Run configurations	25
Setup of access points and HiveManager	28
Dataporten	34
Backend code components	36
Abstract classes	36
ValidatorMixin	36
Authenticator	37
AdHocAdapter	38
lotConnectView	39
Utilities	48
Final classes and utilities	49
Views	49
DataportenRedirectView	49
ConnectView	51
FeideAuthenticator	52
HiveManagerAdapter	53
Utilities	57
Frontend components	59

Markup and styles	59
Brief description of HTML	59
CSS	59
Responsive content	60
Accessibility	61
JavaScript	62
Initial	63
Validation	65
POST request	66
Visual feedback	68
Displaying of PSK	69
Testing and verification	72
Unit tests	72
API tests	74
Class tests	76
Resulting changes	76
Manual testing procedures	76
Verification of proper SSL encryption	77
Testing of encryption between client and access point	78
Testing of roaming	78
Testing of traceability	78
Integration tests	79
GDPR compliance	84
Discussion	85
Future possibilities	85
Admin dashboard	85
User dashboard	86
Custom emails	86
Device type restrictions	86
Serializers as validators	87
WPA3	87
Feasibility	87
Amendment: Admin dashboard	89
Deviations from the design report	99
Unnecessary security feature	99
Email input field	99
Exception cases	99
Checking of support for cookies	100
Conclusion	100
Bibliography	101

Definitions

<i>method</i>	A procedure. ¹
<i>function</i>	A method that returns a value.
<i>frontend</i>	The part of the service that the user interacts with directly.
<i>backend</i>	The part of the service that the frontend communicates with; the part of the service that performs the logic, given that the logic is performed on a remote server that is not usually interacted with directly.

¹ A method that does not belong to a class is usually called a procedure. In this paper, we will not distinguish between procedures and methods. In PyCharm, these three terms are used interchangeably.

Setup of Django

Installation

Our backend was built using Django REST framework, a Python framework for building RESTful APIs. Django REST framework is an extension of Django. We will refer to the collection of Django and Django REST framework as *Django*.

In order to install Django, we installed a package installer, namely *pip*. On the production server, we installed pip with the following command:

```
$ apt install python3-pip
```

This installs pip3, which enables us to install Django:

```
$ pip3 install django    installs Django;  
$ pip3 install djangorestframework  installs the Django REST framework.
```

It is often the case that the testing, staging and production server require different sets of packages to be installed. At the same time, we may work on different projects with wildly different package requirements on the same computer. Therefore, we need a tool that bundles packages into environments that we can switch between. *virtualenv* is such a tool. With it, we can create a virtual environment that we install packages in. If we want to switch out one set of packages for another, we can simply deactivate that virtual environment and activate another. We installed virtualenv using pip:

```
$ pip3 install virtualenv
```

We created a user for our application. This user was given the permissions required to restart relevant services. We named this user *iotconnect*. In its home directory, we created a new directory called *iotconnect*. We also created a virtual environment here named *venv*:

```
$ virtualenv /home/iotconnect/venv
```

We activated this virtual environment using the following command:

```
$ source /home/iotconnect/venv/bin/activate
```

We started a new Django project in the *iotconnect* directory using the following command:

```
$ django-admin startproject iotconnect /home/iotconnect/iotconnect
```

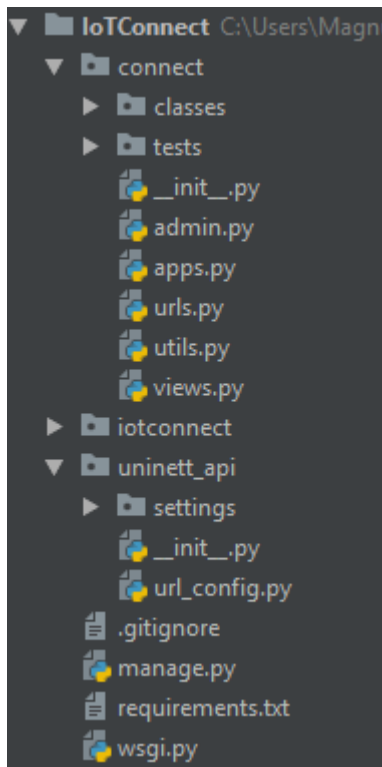
Using Git, we pushed this initial state to a Git repository, which we can pull from after it has been updated. The testing server uses the same directory and user hierarchy.

Code editor

We edited the project's code using the code editor PyCharm by JetBrains. JetBrains offers a free license for students. It is well integrated with Git and provides several useful Git features, such as local code history.

Project structure

We set up our Django project structure in a way that will look familiar to those with experience with working in Django. The structure can be seen in this project explorer:



Apps and URLs

In Django, apps are, simply put, collections of modules (Python files). In the project explorer (see the previous page), you can see several directories. These directories contain an `__init__.py` file (which are usually empty), making them Python packages instead of directories. Packages can be imported in Python code.

What makes a Python package an app, in short, is the presence of a declaration of a subclass of the built in `AppConfig` class within that package. We added the `connect` app, which consists of the `connect` package. In its `apps.py` file, this class declaration can be found:

```
from django.apps import AppConfig
```



```
class ConnectConfig(AppConfig):
    name = 'connect'
```

We made this app an *installed* app by adding it in the base settings file's `INSTALLED_APPS` list:

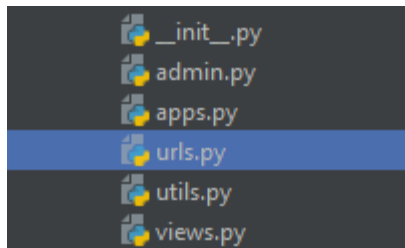
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.postgres',
    'rest_framework',

    # Third-party apps
    'corsheaders',

    # Project apps
    'connect.apps.ConnectConfig',
]
```

This settings file is explained further in the *Hiding secrets and localizing settings* subchapter.

Within the *connect* app's hierarchy, you will also see a file named `urls.py`:



This file contains two URLs, or paths:

```
urlpatterns = [
    path('', ConnectView.as_view(), name='connect'),
    path('complete/dataporten/',
         DataportenRedirectView.as_view(),
         name='dataporten-redirect'),
]
```

The `path` function takes, in our case, three arguments.

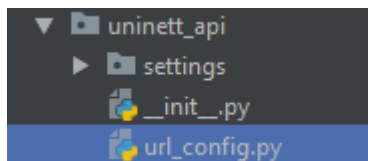
The first argument is the relative path to the endpoint. Notice the second path: “complete/dataporten/”. This is a path that is relative to the base of the app. We will give this

app a prefix (“connect/”) promptly, thus making the second path “host:port/connect/complete/dataporten/” and the first path simply “host:port/connect/”.

The second argument is a class — specifically the *view* class that will handle requests sent to this URL. We will explain these view classes in detail later.

The third argument is the name of this endpoint. This is useful if we want to find the URL of a specific endpoint without knowing the complete path of it. URLs sometimes change, and it would be an inconvenience to have to know every endpoint’s path at all times. We can find the URL of an endpoint using the `reverse(name)` function, which we will be using in unit tests.

We have installed the *connect* app and declared a couple of URLs, but we still need to tell Django where these URLs are located and, indeed, whether or not we want to use them. In the package *uninett_api*, we have created a file called *url_config.py*.



These are its contents:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Project app urls
    path('connect/', include('connect.urls')),
    # Developer tools
    path('admin/', admin.site.urls),
]
```

Here, we again use the `path` function to create URLs. The second one, *admin/*, is the path to the Django admin dashboard. We will touch on this subject later. The interesting path here is the *connect/* URL.

Here, we declare the variable *urlpatterns* to equal a list containing the admin dashboard URL and a *collection* of URLs under the *connect/* URL. The `include` function accepts a collection of URLs. In other words, we are declaring that we want the two URLs declared in *connect.urls* to exist within the “*connect/...*” URL namespace. We could keep adding prefixes in this way for as long as we like, but this will suffice in our case.

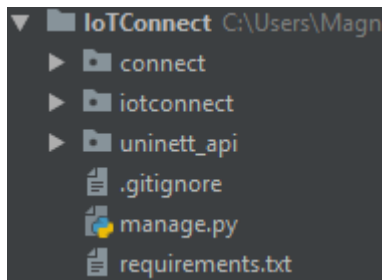
Finally, we must investigate the project’s settings, which are located in the file *uninett_api.settings.base*. Here, we will find this line:

```
ROOT_URLCONF = 'uninett_api.url_config'
```

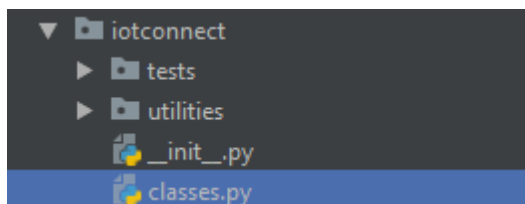
Here, we tell Django (assuming we choose to use *uninett_api.settings.base* for our settings, which we partially will) that the URLs we wish to implement in our project can be found in the file where we declared the *connect/* namespace and included the URLs declared in *connect.urls*.

Other packages

While both the *iotconnect* package and the *uninett_api* package can be found at the same level in the hierarchy as the *connect* app, these two packages are not apps. They do not declare any app URLs or final (not abstract) view classes of their own, and do not contain an *AppConfig*.

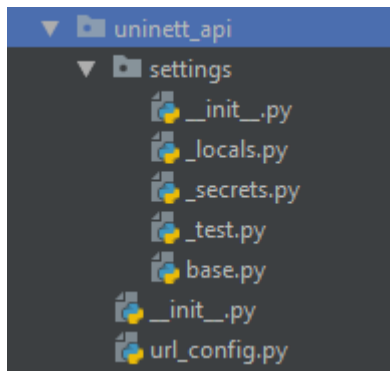


The *iotconnect* package contains the abstract classes to be used by any implementation of this solution. Abstract classes are classes that are meant to be inherited from and not used directly.



We could have chosen to move all of these classes and utilities into the *connect* app, but the *connect* app declares logic that is specific to both HiveManager and Dataporten. Therefore, we separate the general code components from the platform specific code within the *connect* app. In other words, the *connect* app is specific to Uninett's environment, while *iotconnect*, which is the general solution, is not.

The *uninett_api* package, though separated from the *connect* app, is indeed specific to Uninett's environment. It contains the settings for the project as well as its URL configuration.



However, because a Django project may eventually implement any number of apps, it is not wise to combine these two packages into one. We do not want the *connect* app to be an integral part of the Django project. We simply want it to be an extension that can be disabled, enabled, changed or replaced at will. In the future, Uninett may install an app for user management, for example, or a live chat. Those apps would likely also come with URLs and endpoints that are included in the URL configuration.

Setup of Git

Branches

We have decided to set up a flow consisting of three main branches: 1) The development branch (master), 2) a testing branch, and 3) a production branch. The rules for these branches are described in the design report. In summary, we will push to feature branches which are merged into the development branch once all tests have passed and the code has been reviewed. The development branch is merged into the testing branch once we have reached what we deem as a release-worthy state. We perform manual testing on the testing branch until we are satisfied with the quality of the code. The testing branch is then finally merged into the production branch, which is pulled on the production server. This is the version that users will be interacting with.

In order to create these branches, we first *check out* (switch to the version of) the master branch, which currently consists of an “empty” (default state) Django project:

```
$ git checkout master
```

We make sure we have the latest version:

```
$ git pull
```

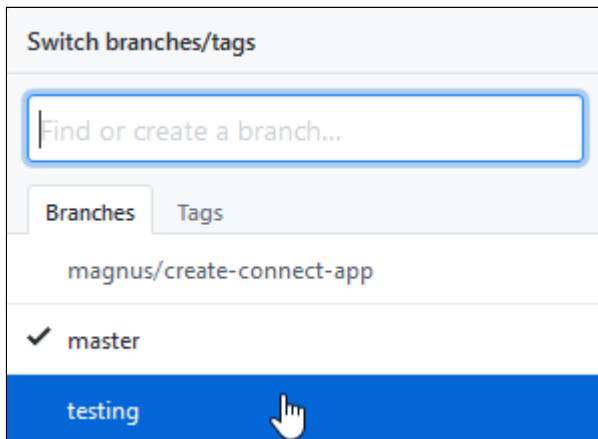
We create a new local branch based on the current (master) branch:

```
$ git checkout -b testing
```

We push the branch to the remote:

```
$ git push origin testing
```

The testing branch can be seen on GitHub:

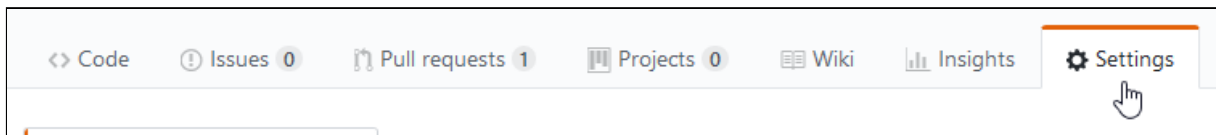


We will do the same for the production branch.

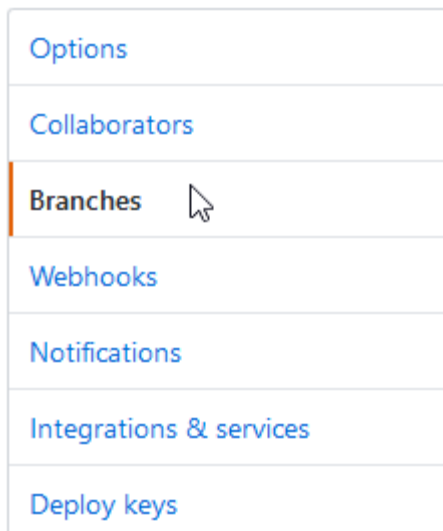
Branch protection rules

Branch protection rules are rules that are set in GitHub in order to protect the integrity and quality of branches. We want to protect our development branch (the master branch) from erroneous and poor code.

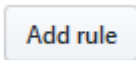
We navigated to the GitHub repository using a web browser and switched to the *Settings* tab:



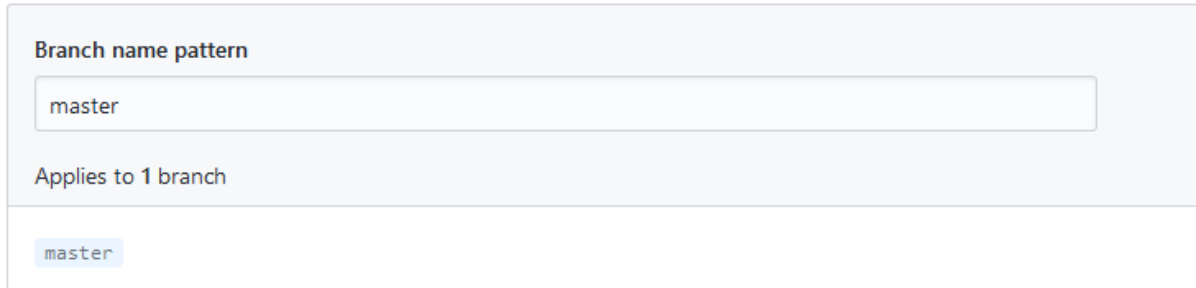
We switched to the *Branches* menu:



We clicked the *Add rule* button:



We chose to apply this rule to the *master* branch:

A screenshot of a web interface for configuring a branch rule. It features a light blue header with the text "Branch name pattern". Below this is a text input field containing the word "master". Underneath the input field, it says "Applies to 1 branch". At the bottom of the form, there is a list of branches with "master" selected and highlighted in light blue.

Here, we checked the option to require pull request reviews before merging. This means that the other developer must read and approve the pull request before it is merged into the master branch. If the developer spots poor code, changes can be requested. We set the required number of approving reviews to 1.

We also checked the *Require status checks to pass before merging* option. Status checks are typically third-party programs that perform some kind of check and prevent one from merging a pull request if these checks fail. We have not implemented such checks at this moment, but plan on using CodeFactor checks. We checked the *Require branches to be up to date before merging* option. This option has the following explanation on GitHub: “This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled.”

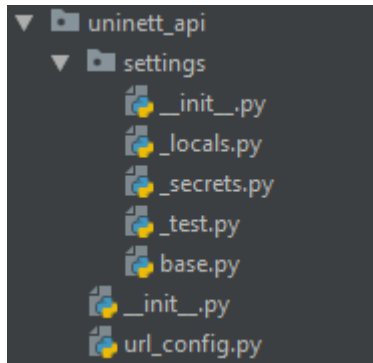
We will not implement status checking for the testing branch or the production branch, as we will not push code directly to these branches. All code will follow the flow from the master branch through the testing branch before ending up in the production branch.

Hiding secrets and localizing settings

Django projects store settings in special settings files. As explained in the design report, our Git repository is public and therefore cannot contain secret keys. These secrets must exist locally only and never be pushed to the remote Git repository. This is achieved using the *.gitignore* file generated by Git, which is a document that tells Git which files should not be included in commits or be regarded as a change that requires a commit.

Typically, cache files and settings pertaining to personal preferences are ignored, as we do not want to enforce one set of preferences for all developers, and we do not want a small change in a cache file to be regarded as a change to the project itself, or indeed to waste storage space and tidiness on such files. But it is also great for keeping secrets local.

Notice that settings we do not want to publish have a leading underscore in the file name:



The ignoring of these settings files is handled by the `.gitignore` file seen at the bottom of the image above. These are the interesting lines within it:

```
*/settings/_*.py
!*/settings/__init__.py
```

The first line says to ignore all files with a leading underscore and ending in `.py`. The second line says that we should not ignore the `__init__.py` file within this directory, even though its name begins with an underscore. This file is required if we want Python to treat these modules as part of a package, which we do.

You will find references to `_secrets.py` and `_locals.py` in our project. If someone wants to clone the repository, they must add these files locally on all computers running the project. This is explained in the *README*² of our repository.

The `_test.py` file contains settings that only apply to this developer's environment. One example of such settings is the name, username and password the be used when Django attempts to connect to the local testing database, which is running on the developer's computer. These variables are user specific. This file also disables certain security mechanisms that are not needed when testing locally. This includes disabling HTTPS and enabling debug mode (extra information returned on exceptions).

This `_test.py` file imports all its settings from the `base.py` file, which is shared among all developers (notice it has no leading underscore). It then overwrites some settings specified in that base file.

Leading underscores were chosen because they are used in Python to make a member appear protected or private. We use the word *appear* because there is no such thing as a truly private member in Python. Importantly, they are also allowed in file names.

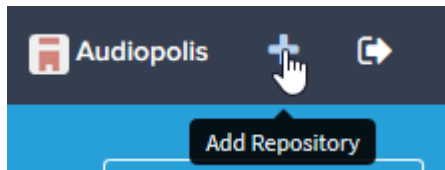
CodeFactor

In the design report, we stated that we plan on implementing CircleCI, a status check for Git repositories that has the ability to automatically run unit tests for us. A pull request would be required to pass these tests before merging. However, we discovered that setting up CircleCI

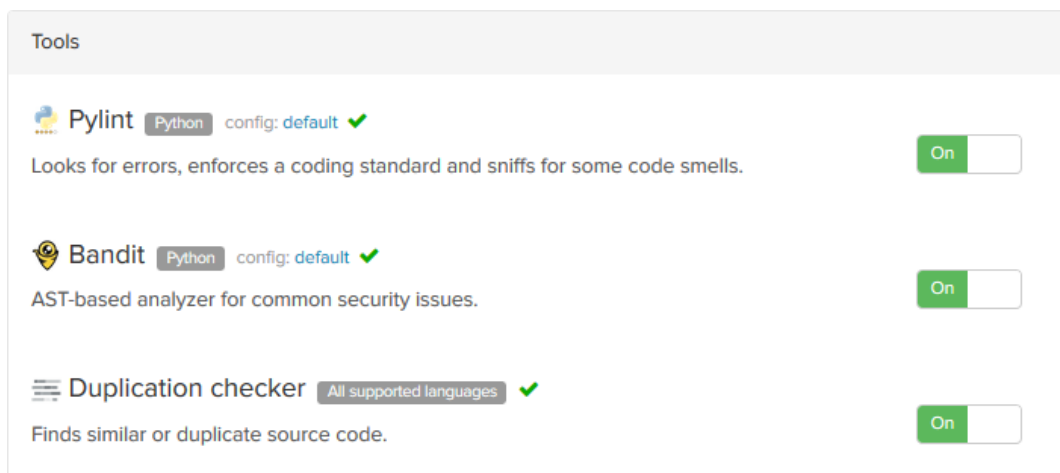
² *README* is a file named `README.md` that is displayed initially at the front page of a repository on GitHub and other Git platforms. Ours is available at <https://github.com/Uninett/loTConnect-BackEnd>.

would be a much too lengthy process compared to the scope of this project. We have very few unit tests and do not frequently push new commits. Therefore, we have decided to save ourselves some time by not implementing CircleCI. Instead, we will add the CodeFactor status check, which checks the syntax and quality of code with minimal setup. We will run the few unit tests we have manually before every pull request.

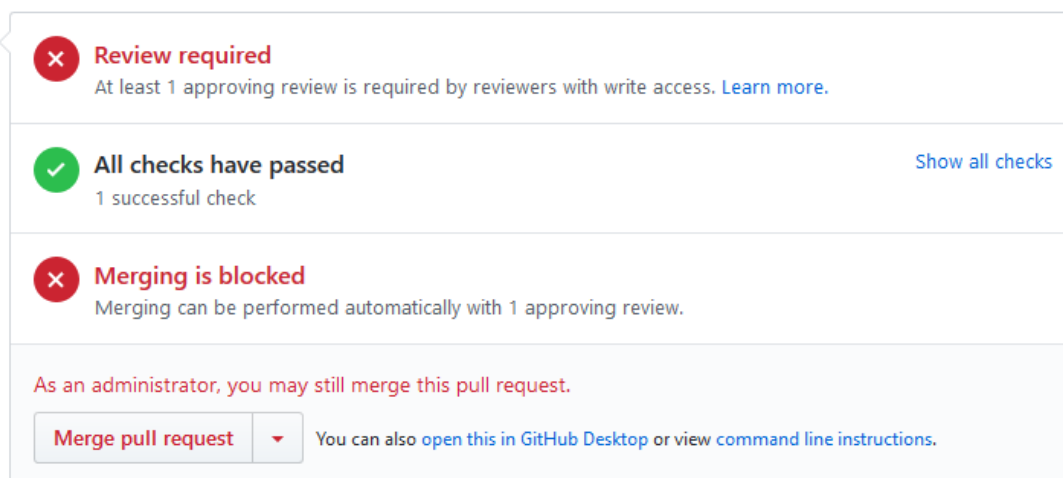
In order to add CodeFactor to our project, we simply need to sign up at codefactor.io and add the repository:



In the settings for this project on codefactor.io, we enabled the *Duplicate checker* tool:



Pylint and *Bandit* are enabled by default. After this, all pull requests will be evaluated by CodeFactor, which looks for issues such as unused variables and imports, missing newlines at the end of files, syntax errors and similar. A passing CodeFactor status check looks like this:



A failed CodeFactor status check looks like this:



If a status check fails, the programmer must address the identified issues, or is not allowed to merge the pull request. Furthermore, the other developer must give an approving review of the changes within the pull request before it can be merged.

Setup of environment

By environment, we mean components that the backend or frontend will interact with, and that are not part of the backend or frontend projects.

Servers

Our testing and production servers are running on virtual machines set up on request by Jørn Åne de Jong at Uninett. We initially had problems getting the Django project to work on these servers due to having used features of Python that are only available in version 3.6. Notable examples include using *f-string*, a way of formatting strings thusly:

```
version: str = "3.6"
f"This only works in version {version} or greater."
```

In Python 3.6 and newer, the above code produces the string "This only works in version 3.6 or greater." Additionally, the type hinting used on the *version* variable (the type of the variable is declared after the colon) is only supported in Python 3.6 and newer. The initially installed version of Debian, however, uses an earlier version of Python. We upgraded our servers' operating systems to a newer version of Debian.

While this got us further, and we managed to get the server to respond back, we needed to install Apache HTTP Server and `mod_wsgi` alongside Django. Apache HTTP Server, commonly called Apache, is open-source web server software that simplifies the process of securing the user's communications with our application using SSL. For Django to be compatible with Apache, we also need `mod_wsgi`, which is a server module that implements a WSGI interface for Python-based web applications. In simpler terms, there are many Python frameworks for web applications out there, and if they are all to be supported by any web server, there must exist an interface between the two that limits the web server's need to be familiar with the individual framework — in our case, Django. That interface is WSGI (Web Server Gateway Interface), which is provided by `mod_wsgi`.

This path would lead to still further problems. `mod_wsgi` needs to be built using the same version of Python as we are using in our project, namely Python 3.6. We downloaded the latest version of `mod_wsgi` from pypi.org (https://pypi.org/project/mod_wsgi/), extracted it and attempted to build it using the *make* utility after configuring³. This, as it turns out, requires the *apache-dev* package, which again had its prerequisites. We also noticed that, after having moved the Git repository from one location to another, the virtual environment we had created for our project's dependencies was pointing at the old, now non-existent location, thus leading to issues with missing packages.

³ A guide can be found at https://medium.com/@garethjohnson_52722/serve-python-3-7-with-mod-wsgi-on-ubuntu-16-d9c7ab79e03a (written by Gareth Johnson on February 6, 2018; retrieved on April 10, 2019).

After spending two days fixing problems like these and more emerging after every fix, we decided that the only remaining options were to either use a Windows server instead, or re-install Linux (preferably Ubuntu). Additionally, we found a more modern alternative to Apache called Caddy. As the website states (<https://caddyserver.com/>) as of April 19, 2019: “Caddy is the HTTP/2 web server with automatic HTTPS.”

For the purpose of installing Caddy, we chose to use Ubuntu Server 18.04 LTS. In order to have more control over our virtual machines, we also chose to use Amazon Web Services instead of virtual machines provided by Uninett.

Installation and configuration of Caddy and gunicorn

Caddy is not itself compatible with WSGI. We achieve compatibility by using gunicorn — a WSGI HTTP server, and by proxying requests from Caddy to gunicorn. gunicorn also enables easy load balancing.

We largely followed a guide (Tuomi, 2018), but deviated from this guide in a few ways.

Caddy can be downloaded at <https://caddyserver.com/download>, where we selected Linux 64-bit as the operating system and added the CORS plugin. We installed gunicorn using Pip:

```
source /home/iotconnect/venv/bin/activate
pip3 install gunicorn
```

The guide explains how to create systemd services for both Caddy and the Django application, but the exact launch command that is used to launch Caddy does not work when called directly by the service (specifically, we were unable to specify the location of the Caddy configuration file, which is `/etc/Caddyfile`). We created a script located in `/home/iotconnect/iotconnect` called `_runcaddy.sh`, which will be executed by a service.

The first service, `iotconnect.service`, has the following contents:

```
[Unit]
Description=IoTConnect Django
After=network.target

[Service]
PIDFile=/run/iotconnect/pid
User=iotconnect
Group=iotconnect
LimitNOFILE=64000
WorkingDirectory=/home/iotconnect/iotconnect
ExecStart=/bin/bash /home/iotconnect/iotconnect/_runserver.sh
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

The `_runserver.sh` script, which is referenced in the above service, contains the following:

```
#!/bin/bash

# Define backend directory
DIR="/home/iotconnect/iotconnect"

# Navigate to the backend directory
cd DIR

# Activate the virtual environment
source ../venv/bin/activate

# Install packages
pip install -r requirements.txt

# Migrate database to latest state
python manage.py migrate --settings=uninett_api.settings._testing

# Start Django with Gunicorn
gunicorn wsgi --bind 127.0.0.1:8000 -w 4 --env \
DJANGO_SETTINGS_MODULE=uninett_api.settings._testing
```

The `gunicorn` command's `--env` option allows us to set the environment variable `DJANGO_SETTINGS_MODULE`, which tells the server where it can find the settings file for the application. For the production server, this path would be `uninett_api.settings._production`. Files with leading underscores are ignored by Git, according to our custom configuration. The `-w` parameter lets us specify the number of workers. Here, we launch four workers running separate instances of the application for load balancing. The `wsgi` argument is the relative path to the `wsgi.py` file, which is found at `/home/iotconnect/iotconnect/wsgi.py`.

The `_runcaddy.sh` script, which we run using another systemd service, contains the following:

```
ulimit -n 8192
caddy -conf "/etc/Caddyfile" -ca \
"https://acme-staging-v02.api.letsencrypt.org/directory"
```

The `ulimit` command limits resource use for users, and the value after the `-n` key specifies the maximum number of open file descriptors. We increase this to 8192, as suggested by Caddy when running it with anything less. We specify the location of the Caddy configuration file: `/etc/Caddyfile`. Lastly, we use Let's Encrypt as our certificate authority. Let's Encrypt issues certificates to pretty much anyone. We use the `acme-staging-v02` sub-domain for this. This will lead to browsers warning the user that the page is not secure, even though the communication is encrypted. But Let's Encrypt's production API (`acme-v02` without the `staging` part) has strict rate limits that will lead to a ban of several days if exceeded. With the staging sub-domain, we can request certificates without worrying about such rate limits. Using the production API on the production server will mean that the site is trusted by most

newer versions of browsers and operating systems. Optionally, we could use a certificate issued by Amazon Web Services (generated using AWS' Certificate Manager), but that would require that we installed the Route 53 (AWS' DNS) plugin for Caddy, which would be a good idea in hindsight, but not worth the trouble. A certificate issued by Uninett is the better option for future development.

The *caddy* service has the following configuration:

```
[Unit]
Description=IoTConnect Caddy
After=network.target

[Service]
PIDFile=/run/caddy/pid
User=iotconnect
Group=iotconnect
RuntimeDirectory=/home/iotconnect
LimitNOFILE=64000
ExecStart=/bin/bash /home/iotconnect/iotconnect/_runcaddy.sh
ExecReload=/bin/kill -s HUP $MAINPID
ExecStop=/bin/kill -s TERM $MAINPID
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
AmbientCapabilities=CAP_NET_BIND_SERVICE
NoNewPrivileges=true

[Install]
WantedBy=multi-user.target
```

We create a third script that takes care of stopping (if necessary) and starting these two services. We called this script *_start.sh*:

```
systemctl stop iotconnect
systemctl stop caddy
systemctl start iotconnect
systemctl start caddy
```

Finally, the Caddy configuration file, */etc/Caddyfile*, has the following contents:

```
api.magnusbakke.com {
    log /home/iotconnect/logs/caddy.log
    errors /home/iotconnect/logs/caddy.log

    proxy / 127.0.0.1:8000 {
        transparent
    }
}
magnusbakke.com {
    log /home/iotconnect/logs/caddy.log
    errors /home/iotconnect/logs/caddy.log
    root /home/iotconnect/frontend
}
```

The log file was created manually by us.

At this point, it is worth mentioning that Magnus registered a domain name (*magnusbakke.com*), which he chose to use instead of the long, complex domain and sub-domain provided by AWS. The production server's domain name, *audiopolis.info*, is also registered by Magnus. This is not an important detail, and we will not give an account of how we registered domain names and used them for this purpose.

In this configuration file, we have defined a "main" domain, and a sub-domain: *api*. The *api* sub-domain is handled by the Django app, while *magnusbakke.com* serves the static frontend files (styled HTML documents with JavaScript). Requests to the *api* sub-domain are proxied to gunicorn, which is serving the Django application at 127.0.0.1:8000. Static files (the frontend) are served normally, and these are located in */home/iotconnect/frontend*. Both are configured to write log entries to stdout (standard output) and error information to stderr (standard error).

Both shell scripts have been made executable using *chmod*.

We run the application thusly:

```
cd /home/iotconnect/iotconnect
./_start.sh
```

Databases

Each server needs a database, including development computers. We have installed PostgreSQL on all servers. On Linux, this is done using the command: `sudo apt-get install postgresql`. All servers, except development computers, are running Linux. On Windows, it is installed using a Windows installer⁴.

Once the database software is installed, we must create a database on each server. Each database should have unique passwords. The password for the database is set in a local settings file that must be created for each server. Postgres was already installed on our development computers running Windows.

We can enter the SQL Shell (*psql*) by switching to the *postgres* user, using `su - postgres`, and starting *psql* using the `psql` command.

Databases are created using `CREATE DATABASE [name];`. We named the database on testing *iot_testing*, and the production database was named *iotconnect_production*.

We also altered the *postgres* database role, giving it a password that is known to the Django application. This is the password we specify in the Django settings in order to gain access to the database:

```
ALTER ROLE postgres password '[redacted]';
```

⁴ Available at <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads#windows>.

These passwords were stored in Django settings files called `_testing.py` and `_production.py` respectively. On development computers, the settings file may be called whatever the programmer likes, but we have chosen to call them `_test.py`. All these settings files are ignored by Git, so the passwords are never exposed to the public. Furthermore, all these settings files extend the base settings, which *are* exposed to the public, as they contain no secret information.

Run configurations

Run/debug configurations are configurations for how a Django server should be run. Each server or development computer will use a different run configuration. The most notable difference between each server's run configuration is the path to the settings file used.

On development computers, we created the run configuration *Test*. The *Test* configuration's environment variables look like this:

Name	Value
PYTHONUNBUFFERED	1
settings	uninett_api.settings._test
DJANGO_SETTINGS_MODULE	uninett_api.settings._test

The local, Git-ignored `_test.py` settings file, which is pointed at by the value of these two environment variables, contains the following:

```
SECRET_KEY = [redacted]
DEBUG = True
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'postgres',
        'NAME': 'iot_development',
        'HOST': 'localhost',
        'PASSWORD': [redacted],
        'PORT': '5432'
    }
}
SECURE_SSL_REDIRECT = False
SESSION_COOKIE_SECURE = False
CSRF_COOKIE_SECURE = False

# Dataporten
SOCIAL_AUTH_DATAPORTEN_FEIDE_SSL_PROTOCOL = False

# Testing variables
```

```
FRONTEND_URL = 'http://iot-backend-tst.labs.uninett.no'  
BACKEND_URL = 'http://127.0.0.1:8000'
```

When running the server using this run configuration, we are able to debug our code locally. API requests are made to 127.0.0.1:8000 from a frontend branch that is configured to make calls to 127.0.0.1:8000 instead of the remote production server.

We set up a similar settings file for the testing server (`_testing.py`), but enabled the security settings related to HTTPS. Each of these servers will have a different value for the `DATAPORTEN_URL` variable in their `_secrets.py` settings file, because each environment (development, testing and production) has a corresponding Dataporten application with a different sign-on URL.

On the production server, we created a run configuration that increases security further by disabling the `DEBUG` setting. This setting, if left enabled, will print secret information to the user upon exceptions. The `_secrets.py` settings files of the testing and production server were also altered to use the Testing application and Production application on Dataporten respectively.

It should be mentioned that run configurations are a concept within the code editor, which does not exist on the testing and production servers. In the case of those servers, by *run configuration*, we mean a command that runs the server using specified settings and flags, such as (in its simplest form):

```
django-admin runserver --settings=uninett_api.settings._production
```

Constructing and executing such commands is essentially what different run configurations do. On the testing and production servers, gunicorn serves the application, which is launched using a different command, as shown in the *Installation and configuration of Caddy and gunicorn* sub-chapter.

This overview shows the main differences between the run configurations and local settings on different environments (secrets have been redacted):

Environment	Setting	Value
Development	DATAPORTEN_CLIENT_ID	857348bd-71b3-443f-be5d-f7bd4421668f
	DATABASES (HOST)	localhost
	SECURE_SSL_REDIRECT ⁵	False
	CSRF_COOKIE_SECURE ⁶	False
	SESSION_COOKIE_SECURE ⁷	False
	FRONTEND_URL	http://iot-backend-tst.labs.uninett.no

⁵ <https://docs.djangoproject.com/en/2.2/ref/settings/#secure-ssl-redirect>

⁶ <https://docs.djangoproject.com/en/2.2/ref/settings/#csrf-cookie-secure>

⁷ <https://docs.djangoproject.com/en/2.2/ref/settings/#session-cookie-secure>

	BACKEND_URL	http://127.0.0.1:8000
	DEBUG ⁸	True
	ALLOWED_HOSTS ⁹	[]
Testing	DATAPORTEN_CLIENT_ID	065fa621-50f7-43a5-b370-ebacb3064c7e
	DATABASES (HOST)	localhost
	SECURE_SSL_REDIRECT	True
	CSRF_COOKIE_SECURE	True
	SESSION_COOKIE_SECURE	True
	FRONTEND_URL	https://magnusbakke.com
	BACKEND_URL	https://api.magnusbakke.com
	DEBUG	True
	ALLOWED_HOSTS	['127.0.0.1', 'localhost', 'api.magnusbakke.com', 'magnusbakke.com']
Production	DATAPORTEN_CLIENT_ID	f2115e09-f47a-4a2f-bd66-f7c9530d7f06
	DATABASES (HOST)	localhost
	SECURE_SSL_REDIRECT	True
	CSRF_COOKIE_SECURE	True
	SESSION_COOKIE_SECURE	True
	FRONTEND_URL	https://audiopolis.info
	BACKEND_URL	https://api.audiopolis.info
	DEBUG	False
	ALLOWED_HOSTS	['127.0.0.1', 'localhost', 'api.audiopolis.info', 'audiopolis.info']

The location of the database is the same for all environments, but only when seen from the perspective of each server. The database is always running locally.

In summary, run configurations are local settings that are not pushed to the Git repository. They determine how the server is run. Each run configuration may specify different Django settings files, which in turn may specify different databases, security settings and more. We created different run configurations for each environment that dictate which database is

⁸ <https://docs.djangoproject.com/en/2.2/ref/settings/#debug>

⁹ <https://docs.djangoproject.com/en/2.2/ref/settings/#allowed-hosts>

used, where the frontend is located, which Dataporten application the user should authenticate for, and more.

Setup of access points and HiveManager

We received two units of Aerohive's AP122, as shown here:



Additionally, Aerohive sent us a license key for HiveManager, and we followed their instructions for creating user accounts.

When configuring access points using HiveManager, it is advised that we first configure a network policy. This is done in HiveManager by signing in and navigating to *CONFIGURE* → *NETWORK POLICIES*:

Here, we found a button labeled *ADD NETWORK POLICY*. When clicking it, we were presented with a form in which we gave the policy a name and a description. We also unchecked the *Switches* and *Routing* policy types, leaving *Wireless* enabled:

What type of policy are you creating?

Wireless
 Switches
 Routing

Please name your policy

Policy Name *

Description

When saving this policy, we are taken to the next tab, which is labeled *Wireless Networks*. We are presented with a button labeled *ADD*:

Wireless Networks

ADD

Assign SSIDs using Classification Rules

<input type="checkbox"/>	SSID
<input type="checkbox"/>	

We chose the option shown below. It is our understanding that the *Guest Access Network* option provides a list of templates, none of which are adequate for our purposes.

ADD

Assign SSIDs using Classification Rules

Guest Access Network

All other Networks (standard)

We gave the new network the SSID and broadcast name *IoT-roam*, and left the *Broadcast SSID Using* options at their default values, as these suit our needs:

Name (SSID) *	<input type="text" value="IoT-roam"/>	Broadcast SSID Using
Broadcast Name *	<input type="text" value="IoT-roam"/>	<input checked="" type="checkbox"/> WiFi0 Radio (2.4 GHz or 5 GHz) <input checked="" type="checkbox"/> WiFi1 Radio (5 GHz only)

We were presented with five options for the type of authentication to be used: 1) *Enterprise*, 2) *Personal*, 3) *Private Pre-Shared Key*, 4) *WEP*, and 5) *Open*. We chose number three: *Private Pre-Shared Key*.

The only option we changed from its default value was the maximum number of clients allowed per PSK, which we changed to 1:

Key Management

Encryption Method

This setting is overridden by the setting in the User Group.

Set the maximum number of clients per private PSK
Range : 0-15, 0=no limit

This setting is only support for local PPSK.

Set the MAC binding numbers per private PSK
Range : 1-5

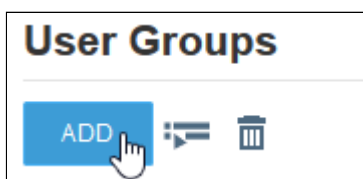
Private Client Group Options

PPSK Classification Options

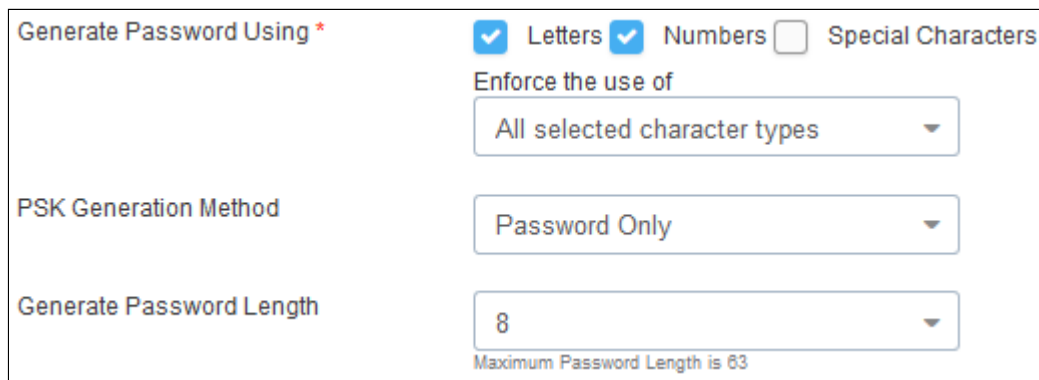
Enable Captive Web Portal

This means that only one client may use a given PSK at any time. We are not able to limit the number of MAC address bindings allowed per PSK, as we are using the cloud-based management system. We do not have a license for a local instance.

On the same page, we are given the option to add a user group. Users in this user group will be allowed to access this network. This is the user group we will be adding users to using the API.



We named the user group *IoT* and gave it the following password settings:



Generate Password Using * Letters Numbers Special Characters
Enforce the use of
All selected character types
PSK Generation Method
Password Only
Generate Password Length
8
Maximum Password Length is 63

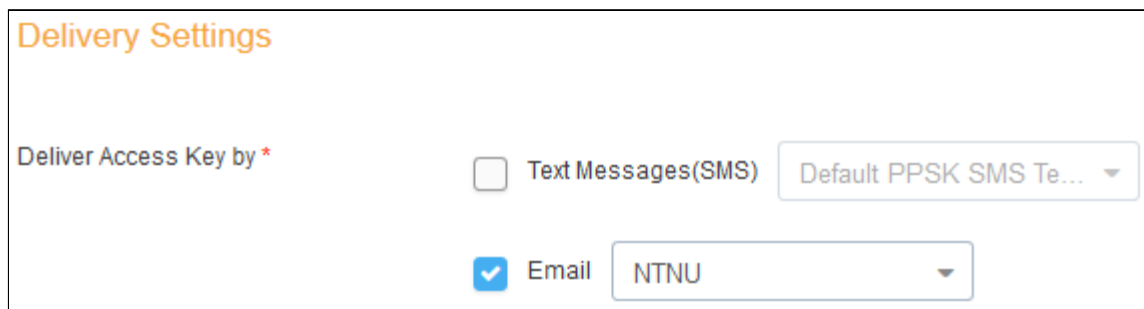
Under *Expiration Settings*, we specified that generated passwords should never expire:



Expiration Settings
Account Expiration
Never Expire

This is a setting that should be discussed.

Finally, we specified the following delivery settings:



Delivery Settings
Deliver Access Key by * Text Messages(SMS) Default PPSK SMS Te...
 Email NTNU

The *Email* field is pointing to a template we created in HiveManager. We will describe the creation of this template shortly. After saving the user group, it appears in the list overview:

<input type="checkbox"/> User Group Name	Password DB Location	# of Users
<input type="checkbox"/> IoT	SERVICE	0 Add

We created a custom template for the email that will be sent to the user when a PSK has been generated. Templates can be created by navigating to *CONFIGURE* → *COMMON OBJECTS* → *BASIC* → *Notification Templates*:

POLICY

BASIC

- Application Sets
- Client Mode Profiles
- DHCP Server and Relay
- IP Objects / HostNames
- MAC Objects / MAC OUIs
- Notification Templates

Here, we added a template by clicking the *ADD* button. We specified the following options:

Name *	<input type="text" value="NTNU"/>
Template Type	<input type="text" value="Email"/>
Security Type	<input type="text" value="PPSK"/>
Icon URL	<input type="text" value="https://i.imgur.com/nBwtkaa.png"/> Recommended size: 110 x 110 pixels
Logo URL	<input type="text" value="https://"/> Recommended size: 300 x 100 pixels
Description Text *	<input type="text" value="You may use this access key to connect to the IoT-roam wireless network. Please note that the access key is only valid for one device."/>
Insert Variable	<input type="button" value="SSID"/> <input type="button" value="LINK"/> <input type="button" value="Preview"/>

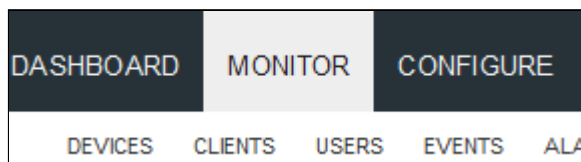
The *Icon URL* points to an NTNU logo¹⁰, which has been uploaded to imgur.com. When a PSK is delivered by email, the email looks like this:



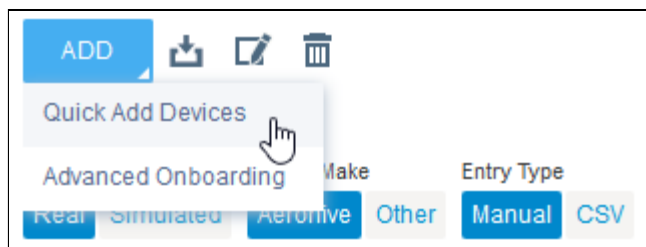
You may use this access key to connect to the IoT-roam wireless network. Please note that the access key is only valid for one device.

Please note that there may be inconsistencies in the appearance of this email in this report, as we have made changes. The above image shows the final version of the template.

Before a working configuration was achieved, we also had to add the access points. By adding their serial numbers of the access points, they will be automatically configured. We navigated to *MONITOR* → *DEVICES*:



Here, we added a new device using the *Quick Add Devices* option:



We entered the serial number of both access points (separated by a comma). We have redacted these serial numbers:

¹⁰ Retrieved from <https://innsida.ntnu.no/logo-og-maler>.

Device Type			Device Make			Entry Type			XXXXXXXXXXXXXXXXX1, XXXXXXXXXXXXXXXXXXXX2		
Real	Simulated	Aerohive	Other	Manual	CSV						

We chose the *IoT-roam* policy under *Add policy*:

Finally, we clicked *ADD DEVICES*. This concludes the configuration of HiveManager and the access points.

Dataporten

Dataporten's dashboard (<https://dashboard.dataporten.no>) allows users to register applications for Dataporten/Feide authentication support. We signed in using our own Feide credentials and registered three applications.

We need one Dataporten application for each environment: Development, testing and production. This is because of the use of redirect URIs in OAuth 2.0. The redirect URI is the URI the user is redirected to after successful authentication with the identity provider. In our case, this is the view at *host/connect/*. In the development environment, the host is 127.0.0.1:8000. On testing, it is magnusbakke.com. On production, it audiopolis.info.

Each application is given its own client ID and secret key, which is part of the reason we need to keep the settings within *_secrets.py* local on the backend, even if the repository was private: Each environment will have different secrets.

There are three pieces of user data we need in our application: 1) The user's email address, 2) the user's name, and 3) the user's Feide username. We added these *scopes* in the *permissions* menu of the Dataporten dashboard, and removed the scopes we do not need:

Accepted scopes

E-post Brukerens e-postadresse ✖ Remove scope	email
ProfilInfo Brukerens navn og profilbilde. ✖ Remove scope	profile
Felde-navn Brukerens identifikator i Feide. Tilsvarende eduPersonPrincipalName. ✖ Remove scope	userid-feide

Backend code components

We developed our code components in close approximation to the designs laid out in the design report. As expected, the actual implementation called for a few additional methods. We will now walk you through the code that constitutes the classes involved in this project. When displaying code, we have lowered the font size in certain places in order to make the entirety of the code more readable. We have also used some syntactic freedom for the same purpose.

We feel that it is necessary to describe the logic being performed in great detail, as this logic is the heart of the solution. It is responsible for its security and the majority of its functionality.

Abstract classes

These abstract classes can be found in the *iotconnect* package. They are meant to be inherited from and have their methods overridden by platform specific code.

ValidatorMixin

ValidatorMixin is a mixin we wrote for use in both the Authenticator class and the AdHocAdapter class. Mixins are classes that are inherited from along with the base class. In Python, a class that does not inherit from any other class will actually inherit from *object*, the base type that all classes must inherit from. It is worth mentioning that there is no such thing as *the* base class when inheriting from multiple base classes, such as a base class and a mixin. The distinction is made by the developer, and which base class becomes the mixin depends on the purpose and scope of each base class.

ValidatorMixin consists of the following code:

```
class ValidatorMixin:
    def validate_data(self, data, **kwargs) -> dict:
        raise NotImplementedError("Override validate_data")

    def get_validated_data(self, data, **kwargs):
        validated_data = self.validate_data(data, **kwargs)
        if validated_data is None:
            raise ValueError("validate_data should return the
                               validated data")
        return validated_data
```

The class contains two functions. The first, `validate_data`, accepts some data to validate and an unknown set of *keyword arguments*. Keyword arguments are received in the form of a dictionary with keys and values, and can either be passed into the function using such a dictionary, or by specifying the name of the argument, followed by an equals sign, followed by the value, like so:

```
return some_function(argument1, argument2, keyword1=some_value,
                    keyword2=some_other_value)
```

We do not presume to know exactly what options and flags the programmer may want to accept and process within these functions. Therefore, we make room for keyword arguments in addition to the mandatory *data* argument.

If called, `validate_data` will throw an exception — specifically a *NotImplementedError* — which is raised (and not handled). This makes sure that the class is not used as is by the programmer. If the programmer attempts to use a standalone `ValidatorMixin` object, an error will occur when `validate_data` is called. The purpose of the class is to be implemented in another class, which is supposed to override the method (write its own version of it that actually performs validation).

The second function, `get_validated_data`, calls the first function and verifies that some validated data was returned. If nothing was returned, the implementation of `validate_data` is wrong; other classes expect the data passed in to be validated and returned if valid. This is crucial. Therefore, a *ValueError* is thrown if no data is returned. This is the very reason why there are two functions. One function ensures that the other returned something. `validate_data` is will not be called directly. The only location where `validate_data` is called is from within `get_validated_data`. The other classes in our solution exclusively call `get_validated_data`.

To summarize, subclasses of both `AdHocAdapter` and `Authenticator` are expected to have the function `get_validated_data`. Because we 1) do not want to repeat ourselves, and 2) cannot trust that the programmer remembers to return the validated data after validating, we created a mixin that is implemented by `Authenticator` and `AdHocAdapter`. This mixin requires that the programmer of subclasses of `Authenticator` or `AdHocAdapter` overrides the `validate_data` function and returns the validated data. This class exists solely to make the task of developing a custom solution easier.

Authenticator

This is a simple abstract class consisting of the following code:

```
class Authenticator(ValidatorMixin):
    request = None

    def is_authorized(self, validated_data) -> bool:
        raise NotImplementedError("Override is_authorized")

    def authenticate(self, authentication_data, **kwargs):
        validated_data = self.get_validated_data(
            authentication_data,
            **kwargs)
        return self.is_authorized(validated_data)
```

Notice that the class inherits from `ValidatorMixin`. This means that the class in fact has four — not two — methods (besides those implemented by the *object* type). These are `get_validated_data`, `validate_data`, `is_authorized`, and `authenticate`. Notice that we have not overridden `validate_data`, as is required by `ValidatorMixin`. This is because `Authenticator` is *also* an abstract class that is not meant to exist as a standalone instance. If we did override `validate_data` here, the programmer would never be told that they need to override it themselves.

The `Authenticator` class has two unique functions: `is_authorized` and `authenticate`. The first, `is_authorized`, is meant to be overridden, as can be deduced by the raising of a `NotImplementedError` should the function be called on an instance of `Authenticator`, or on a subclass of `Authenticator` that does not override the function. This function is supposed to contain the logic that determines whether or not the user should be allowed to proceed to generate a PSK. In our case, this logic will consist of checking if the user has authenticated using Dataporten's single sign-on page.

The second function, `authenticate`, first calls the `get_validated_data` function of the `ValidatorMixin` and stores it in a variable. Here, you can see the importance of the validated data actually being returned. If the authentication data (credentials, for example) was validated but not returned, then `is_authorized` would receive an empty variable (`NoneType`), and it would be unable to determine whether or not the user should be allowed to generate a PSK. Without the raising of an exception, the programmer would have a hard time figuring out why the solution isn't working.

`authenticate` is not meant to be overridden.

Additionally, the class has a `request` attribute, which is set by a different class when a request arrives at the backend. This makes the original request available if necessary for authorization.

AdHocAdapter

This abstract class is equally simple. This is the code it consists of:

```
class AdHocAdapter(ValidatorMixin):
    request = None

    def perform_generation(self, validated_data) -> Response:
        raise NotImplementedError("Override perform_generation")

    def generate_psk(self, generation_options, **kwargs):
        validated_data = self.get_validated_data(
            generation_options, **kwargs)
        return self.perform_generation(validated_data)
```

This class also implements the `ValidatorMixin`, thus giving it four functions (excluding those implemented by the *object* type): Those declared within `ValidatorMixin`, and those declared within this class.

Similarly to `Authenticator`'s `validate_data`, this class has an abstract `perform_generation` function, which must be overridden. It accepts the validated data returned by the authenticator. This method is supposed to perform the custom logic required to generate the PSK, which in our case means calling an API endpoint of `HiveManager`.

Additionally, it has a `generate_psk` function, which is not meant to be overridden. It gets the validated data by calling the `get_validated_data` function of `ValidatorMixin`, which it then passes to the `perform_generation` function as an argument. In this way, the validated data is accessible to the programmer from within the overridden `perform_generation` function.

This class also has a *request* attribute for the same reasons as the `Authenticator` class. `perform_generation` is expected to return a `Response` (Django REST framework's built in response class) object, as is indicated by the "`-> Response`". This is not a hard rule. Different implementations of Django may use different response classes. Therefore, we do not enforce this rule programmatically, but the code editor should display a warning to the programmer if they return an object that is not a `Response` object. This is the response that is returned to the user, and it might, for example, contain the PSK itself (if this is desirable).

lotConnectView

`lotConnectView` is by far the most extensive class in this project. This is the base class for views that receive PSK requests, call the `Authenticator` and generate the PSK. Views that inherit from this class act as the *receiver* component in our designs.

In the following code, we have collapsed the functions, thus hiding their logic. We will first provide an overview of the class attributes before looking at each function (declared using the keyword *def*) in detail separately.

```
class IotConnectView(APIView):
    ad_hoc_adapter: AdHocAdapter = None
    authenticator: Authenticator = None
    requires_authentication: bool = True
    authentication_data = None
    generation_options = None
    permission_classes = []
    generation_method = IotRequestMethod.POST
    authentication_failed_redirect_uri = None

    def _validate_attributes(self): ...

    @staticmethod
    def _validate_request(request): ...
```

```
def _handler(self, request, **kwargs): ...

def dispatch(self, request, *args, **kwargs): ...
```

The class inherits from Django REST framework's built in `APIView` class, which is thoroughly described in the documentation¹¹.

The first two class attributes, `ad_hoc_adapter` and `authenticator`, are meant to be set by the programmer when writing a custom view. As indicated by the type hints, these are expected to be instances of subclasses of `AdHocAdapter` and `Authenticator`.

The third attribute, `requires_authentication`, defaults to `True`, but can be set to `False` by the programmer to explicitly tell the solution to ignore the absence of an authenticator. If set to `True`, the code within `_validate_attributes` will not raise an exception if `authenticator` is `None`.

`authentication_data` and `generation_options` hold the values of the `authentication_data` and `generation_options` keys in the request (which are mandatory) once they have been extracted from the request data. These variables are not used directly in this class, but may be useful to the programmer developing a custom solution.

`permission_classes` is an attribute in the `APIView` base class. It is expected to be a list of permission classes, which are classes that perform checks and determine if the user should be allowed to perform the request or not. Because we want our solution to work with existing solutions that implement their own permission classes, we have decided not to make `Authenticator` a permission class. By default, `lotConnectView` subclasses will not perform any permission checking. This can be changed by the programmer in a custom solution by explicitly setting these permission classes.

`generation_method` is expected to be a value of the `lotRequestMethod` enumeration. An enumeration, or *enum*, is a finite set of possible choices. The possible choices here are 'GET', 'POST', 'PUT' and 'PATCH', which are all HTTP method names. A request coming in can use any of these (and a few more) methods, and if the method of the incoming request equals the value of this variable, the request will be regarded as a PSK request, and the built in handler will be called. The variable defaults to `lotRequestMethod.POST`. The programmer should not write a method called `post` if the value of this variable is also `POST`. This goes for all HTTP methods. This will become clearer after having gone through the logic in the various methods.

Finally, `authentication_failed_redirect_uri` can optionally be set in order to redirect the user if authentication fails. This will not be used in our case, but may be useful for a different custom solution.

Moving on to the functions in this class, let us say that the order in which they are declared is not important. The order shown above does not necessarily reflect the order in which they are declared in the source code. While we would choose a declaration order that puts the "private" methods (with a leading underscore in the name) at the bottom and the rest in the

¹¹ Available at <https://www.django-rest-framework.org/api-guide/views/>.

order in which they are called, we chose the order seen above for this paper because it allows us to explain what a certain method does before it is called by another function.

The first function, `_validate_attributes`, accepts no arguments. The `self` argument is automatically passed when calling instance methods (as opposed to class methods or static methods), and is a reference to the object that the function is called on. The `self` object allows us to access the attributes and other methods of the object.

This is what the `_validate_attributes` function looks like:

```
def _validate_attributes(self):
    if not self.ad_hoc_adapter:
        raise AttributeError("The attribute 'ad_hoc_adapter'
                               must be set.")
    if not self.authenticator and self.requires_authentication:
        raise AttributeError("The attribute 'authenticator' must
                               be set if 'requires_authentication'
                               is True.")
    if not hasattr(self.ad_hoc_adapter, 'generate_psk'):
        raise AttributeError("The ad hoc adapter used should
                               inherit the AdHocAdapter and
                               implement all abstract methods.")
    if self.authenticator and not hasattr(self.authenticator,
                                           'authenticate'):
        raise AttributeError("The authenticator used should
                               inherit the Authenticator class and
                               implement all abstract methods.")
```

When called, the function performs a series of checks that determine whether or not to throw an exception. When an exception is thrown, it means an error occurred. Such errors must be handled by the calling code (or any code farther down the call stack), or they will disrupt the expected behavior of the program. Such disruptions are made visible to the programmer, so that they know it occurred and can do something about it. An unhandled exception causes the subroutine to break, so the following code will not be executed.

This function first asserts that the `ad_hoc_adapter` attribute is set on the view. This is the component that generates the PSK and is therefore an integral, mandatory attribute that must be set. If it is not set, an *AttributeError* will be thrown with the following message: "The attribute 'ad_hoc_adapter' must be set."

If it is set, it will check if the `authenticator` attribute is set. If it is not set, and the view's `requires_authentication` attribute is also `True`, an exception will be thrown with the following message: "The attribute 'authenticator' must be set if 'requires_authentication' is `True`." Not all solutions will require any form of authentication. Therefore, we must check if the view is configured to not require authentication.

Next, the function checks if the `ad_hoc_adapter` object has an attribute called `generate_psk`. The source code later expects that the ad hoc adapter has a method called

`generate_psk`. The attribute is presumed to be callable (it could in theory be anything from a callable function to a number). If it is not callable, an exception will be raised once the code attempts to call it as if it was a function.

Finally, the function checks if the `authenticator` attribute is set. As you will recall, we already did this check, but keep in mind that there is a scenario in which the authenticator will *not* be set (namely if it is not set and `requires_authentication` is also `False`). Therefore, we again have to check if it is set. If it is set, the function checks if the authenticator has an attribute named `authenticate`, as the existence of such a method is later expected. If it does not have said method, an exception is thrown with the following message: "The authenticator used should inherit the Authenticator class and implement all abstract methods."

As mentioned in the design report, we do not wish to enforce the usage of the `AdHocAdapter` class or the `Authenticator` class, as long as whatever custom class used instead has these required functions. Therefore, we do not explicitly check if the values of these attributes are instances of subclasses of these classes.

The `_validate_request` method is a *static* method. Static methods are methods whose logic does not depend on any instance attributes, which may vary depending on which instance of a class you inspect. Static methods perform the same logic no matter which instance they are called on. You will notice that the function never accesses the *self* argument, and that the function indeed does not accept such an attribute:

```
@staticmethod
def _validate_request(request):
    try:
        authentication_data = request.data['authentication_data']
    except KeyError:
        raise ValidationError("authentication_data is required.")

    try:
        generation_options = request.data['generation_options']
    except KeyError:
        raise ValidationError("generation_options is required.")

    return authentication_data, generation_options
```

`@staticmethod` is a *decorator*. In Python, decorators are placed just before a function or class declaration with a leading ampersand (`@`). Decorators, simply put, are functions that wrap other functions. Without going into further detail, we can simply say that the `@staticmethod` decorator tells the function that the first argument (which is usually called *self* in instance methods) is in fact *not* an instance of the class. The function would still work if we removed the decorator and added an initial *self* parameter but, as we never access instance attributes or methods inside this function, it would be redundant, and PyCharm — the code editor used in this project — would display a notice saying “this method may be static.”

The function expects an object. The parameter is named *request*. There are several request classes out there, and we do not make any requirements of this object except for the support of the methods and attributes called and accessed.

First, the function attempts to retrieve the value of the *authentication_data* key in the request data. The *data* attribute of the request is a dictionary. Dictionaries have keys with values, and the values of given key/value pair is retrieved using brackets containing the key. If accessed in this way, a *KeyError* will be thrown if the key is not present in the dictionary. If so, the function throws a more fitting *ValidationError* with a message that explains what is required of the request data. Unless handled (which our code does not), Django will handle the exception by returning a Bad Request response with the provided message to the user.

The exact same check is made for the expected *generation_options* key.

If both keys are present, the two values are returned. In Python, we are allowed to return multiple values in this way, because separating objects with a comma automatically constructs a tuple (a list of constant length) for us.

The `_handler` method deals with some of the methods described thus far.

```
def _handler(self, request, **kwargs):
    # Ensure the presence of required data fields
    self.authentication_data, self.generation_options = \
        self._validate_request(request)

    # Call the authenticator and return a Forbidden response if
    # authentication fails
    if self.requires_authentication:
        if not self.authenticator.authenticate(
            self.authentication_data,
            **kwargs
        ):
            if self.authentication_failed_redirect_uri is not None:
                return redirect(
                    self.authentication_failed_redirect_uri
                )
            return Response(status=status.HTTP_403_FORBIDDEN)
    # Generate the PSK
    response = self.ad_hoc_adapter.generate_psk(self.generation_options,
                                                **kwargs)
    return response
```

Please note that backslashes (\) are used when a statement is too long for a single line. The statement is continued on the next line.

The function accepts a request argument and an arbitrary number of *keyword arguments*. These are contained within the ***kwargs* argument, which are not used in our solution, but may be used in other custom solutions.

First, the function calls the `_validate_request` function and stores the two returned values in equally named instance attributes. In other words, it is given two dictionaries from

`_validate_request`, which it then stores in the `authentication_data` and `generation_options` attributes of the `self` object (the instance of this class that the method was called on). Instance attributes (`self.something`) may be accessed by other functions at a later time. If any exception was thrown (possibly a `ValidationError`), the exception will be raised (as it is not handled using a `try/except` clause), the execution of the method will stop, and a `Bad Request` response will be returned.

Next, the method checks if authentication is required (as determined by `requires_authentication`). If so, it calls the `authenticate` method of the authenticator, passing the request and keyword arguments to it. If `authenticate` returned `False`, the method checks if `authentication_failed_redirect_uri` is set. If so, a `redirect` response is returned, redirecting the user to that URI. Otherwise, a `Forbidden` response is returned.

If `authenticate` returned `True`, the `generate_psk` method of the ad hoc adapter is called. Again, the request and keyword arguments are passed to it. `generate_psk` is expected to return a response, which is finally returned by this function.

Finally, the `dispatch` function must be addressed. `dispatch` is a function that is declared in the `APIView` base class. We have overridden it, meaning that our version of the function will be called instead when using the `lotConnectView` class. This function is called once a URL has been resolved (meaning the appropriate view handling the URL has been found). It is responsible for finding the appropriate method handler within the view, setting some initial attributes and similar. This is what it looks like:

```
def dispatch(self, request, *args, **kwargs):
    """
    Overridden to validate request data and call the
    authenticator and PSK generator's methods.
    """
    # Set attributes and initialize request (as done in the base method)
    self.args = args
    self.kwargs = kwargs
    request = self.initialize_request(request, *args, **kwargs)
    self.request = request
    self.headers = self.default_response_headers

    # Ensure that required view attributes are set
    self._validate_attributes()
    # Set the request attribute on the authenticator
    if self.authenticator:
        self.authenticator.request = request
    # Set the request attribute on the adapter
    self.ad_hoc_adapter.request = request

    setattr(self, str(self.generation_method.value), self._handler)

    try:
        # Get the handler
        handler = getattr(self, request.method.lower(),
```

```

        self.http_method_not_allowed)
    # Call super's initial
    self.initial(request, *args, **kwargs)
    # Get the response
    response = handler(request, *args, **kwargs)
except Exception as exc:
    response = self.handle_exception(exc)

self.response = self.finalize_response(request, response, *args,
                                       **kwargs)

return self.response

```

This function is largely similar to the overridden method in the base class, but performs some additional checking. Please note that the text encapsulated by three quotation marks on each side is called a *docstring*, and is used as an explainer for programmers. It does not affect the logic within the function. We should also mention that lines with a leading # are comments, which also do not affect the logic.

First, some instance attributes (namely `args` and `kwargs`) are set. This is done in the base method. We will not explain these attributes as we will not use them.

Next, the request is converted into an instance of Django REST framework's Request class, which is also done by a method in the base class called `initialize_request`. This converted request object is stored in the view's `request` attribute. The default response headers are then set. This is also taken from the base class, and we will not go into further detail.

After this initial setting of attributes, the function calls the `_validate_attributes` function. This function, as you may recall, ensures that the required attributes (namely `ad_hoc_adapter` and possibly `authenticator`) are set.

Next, provided that the `authenticator` attribute is set, the authenticator's `request` attribute is set with the REST framework Request instance. The same is done for the ad hoc adapter. This makes the request easily accessible to these instances in case they are necessary in a custom solution.

The following requires some explanation:

```
setattr(self, str(self.generation_method.value), self._handler)
```

Python is a highly dynamic language that allows us to set attributes that do not exist yet. The default value of `generation_method` is `HttpRequestMethod.POST`, which has a corresponding string value of "post". While the `self` object (this view) does not have an attribute/method named `post`, we can give it an attribute named `post`. `setattr` accepts three arguments: 1) The object that should be given an attribute, 2) the name of the attribute (which may or may not already exist), and 3) the value of the attribute. Attribute values can also be callable methods.

Here, assuming the programmer has not changed the default value of `self.generation_method`, the statement causes the current view to be given a new

attribute named *post* with a value of the callable method `_handler`, which we have already described. The result is that we may now do the following:

```
response = self.post(some_argument)
```

If we did not need our views to support more than a single HTTP method, we could simply call the handler right now and return its value, but we need to support at least two different HTTP methods (POST and GET). The reasoning here will become clearer by the end of this chapter.

The following code also needs a more thorough explanation:

```
try:
    # Get the handler
    handler = getattr(self, request.method.lower(),
                      self.http_method_not_allowed)
    # Call super's initial
    self.initial(request, *args, **kwargs)
    # Get the response
    response = handler(request, *args, **kwargs)
except Exception as exc:
    response = self.handle_exception(exc)
```

In the *try* clause, we use `setattr`'s soul mate, `getattr`, in order to get an attribute. We cannot know in advance whether or not a subclass has a method named *get* or *patch* (both valid HTTP methods), for example, and we also don't know in advance which HTTP method will be used. Therefore, we need to access the attribute (which may or may not exist) using a dynamic method.

If we assume that the request's method is *POST* and that the value of the `generation_method` is `HttpRequestMethod.POST`, that means (thanks to `setattr`) that this `getattr(self, request.method.lower())` will return the value of the `post` attribute, which was set to be the `_handler` method. If the request method is *GET*, however, `getattr(self, request.method.lower())` may throw an exception, because this class has no `get` attribute. It is possible that a subclass has declared a `get` attribute, and in that case, the statement will return that method.

We do not want to throw an exception when an unsupported HTTP method is used. The standard response is a 405 Method Not Allowed response. Luckily, `getattr` lets us set a default return value in case the attribute does not exist, and the base class, `APIView`, has a method that returns such a response. We will use this method as the default return value.

Let us assume that the value of the `generation_method` attribute is `HttpRequestMethod.POST`, and that the subclass (which inherits from this class) has defined a `get` method, then we can give a few examples of what will be returned given various HTTP methods:

<code>request.method.lower()</code>	<code>getattr(self, request.method, self.http_method_not_allowed)</code>
<code>"post"</code>	<code>self._handler</code>
<code>"get"</code>	<code>self.get</code> (declared in subclass)
<code>"patch"</code>	<code>self.http_method_not_allowed</code>
<code>"put"</code>	<code>self.http_method_not_allowed</code>
<code>"delete"</code>	<code>self.http_method_not_allowed</code>

All of these return values are valid handlers.

Next, the `initial` method of `APIView` is called. This method performs some logic that is not relevant to our solution, but will be in most cases. Among other things, it checks if the user has permissions to use the endpoint using Django's built in permission system.

We now certainly have a handler. This handler may or may not be `http_method_not_allowed`. Whatever the case, as a last step in the `try` clause, the function calls the handler and passes the request plus any arguments and keyword arguments into it. The returned value is expected to be a response.

If any exception was thrown during any of these steps, the `except` clause will be triggered. This is the broadest possible `except` clause:

```
except Exception as exc: ...
```

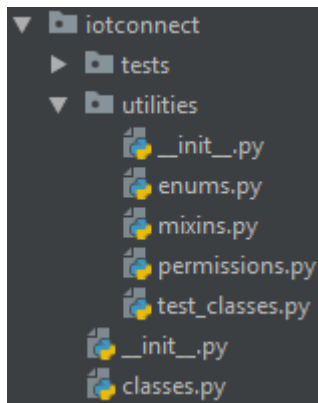
Its broadness comes from the fact `Exception` is the most basic exception class, and all other exceptions are descendants of it. In other words, all possible exceptions are caught by this clause.

The `except` clause calls the `handle_exception` method of `APIView`, passing the exception object. If Django has some default way of handling the exception, an appropriate response (with an appropriate HTTP status code, such as 500 Bad Request) will be returned. If Django has no default way of handling the exception, the exception will be re-raised (and not handled).

Finally, after the `try/except` clause, the `finalize_response` method of `APIView` is called. We will not go into detail about its logic, but this method may optionally be overridden by the programmer if some final modifications must be done to the response object before it is returned to the user, for example. The returned object of this function is the final response, and is stored in the `self.response` attribute before being returned to the user.

Utilities

The *iotconnect* package contains a package called *utilities*, which contains four modules:



The first module, *enums.py*, contains the declaration of the `IotRequestMethod.POST` enumeration:

```
from enum import Enum

class IotRequestMethod(Enum) :
    GET = 'get'
    POST = 'post'
    PUT = 'put'
    PATCH = 'patch'
```

Notice that we left out several HTTP methods. We did this simply because using the DELETE or OPTIONS method, for example, for *generating* PSKs would be ridiculous.

Enumerations are used when we want to limit the freedom of choice in some situation. Here, we want the programmer to pick one of these four HTTP methods to be used for generating PSKs. The default is `IotRequestMethod.POST`, because the POST method is typically used for creating new objects based on some input data, which is the case in our solution. A custom solution that does not require any authentication data or generation options would probably use the GET method instead, as it seems to the user as if they are simply retrieving a password.

The contents of *mixins.py*, which only contains the declaration of the `ValidatorMixin` class, has already been described.

The contents of *permissions.py* is irrelevant to our solution. It contains the declaration of the default permission class (as specified in the base settings file). All our views specifically set the `permission_classes` attribute using an empty list, rendering the default permission class (which raises an exception saying that permissions are not set) irrelevant.

The *test_classes.py* module contains a test class that simplifies the process of writing unit tests. We will go into further detail about this class in the *Testing and verification* chapter.

Final classes and utilities

Views

The Feide+HiveManager solution incorporates two final view classes: 1) The `DataportenRedirectView` class, and 2) the `ConnectView` class.

In the file `url_config.py`, as we have already, includes the following declaration:

```
urlpatterns = [  
    # Project app urls  
    path('connect/', include('connect.urls')),  
  
    # Developer tools  
    path('admin/', admin.site.urls),  
]
```

The `'connect.urls'` string points to the `urls.py` file in the `connect` app, which contains the following:

```
urlpatterns = [  
    path('', ConnectView.as_view(), name='connect'),  
    path('complete/dataporten/', DataportenRedirectView.as_view(),  
         name='dataporten-redirect'),  
]
```

Together, this means that we have two URLs (excluding the admin site): `hostname/connect/` and `hostname/connect/complete/dataporten/`. The former points to an instance of `ConnectView`, and the latter points to an instance of `DataportenRedirectView`. The end of the former — `/complete/dataporten/` — is presumably required by the `python-dataporten-auth` package, which we originally attempted to use before concluding that it is not necessary in this case.

DataportenRedirectView

The `DataportenRedirectView` is the callback that the user is redirected to after successful authentication with Feide/Dataporten. When the user is redirected, a code is sent as a query string (`hostname/connect/complete/dataporten/?code=[something]`). This code is the key our application needs in order to retrieve an access token from Dataporten, which in turn may be used to retrieve personal information about the user who just authenticated themselves.

The `get` function makes use of a few utilities, such as `urlconf` and `urlresolvers`, which will be explained at the end of this subchapter.

```
class DataportenRedirectView(APIView):  
    permission_classes = []  
  
    def get(self, request, **_kwargs):
```

```

code = request.query_params.get('code', None)

if not code:
    raise ValueError("code was not supplied")

access_token = get_access_token(code)
user_data = get_user_data(access_token)
request.session['access_token'] = access_token
request.session['user_data'] = user_data

if not request.session.session_key:
    request.session.save()

session_key = request.session.session_key
name = get_first_name(user_data)
query_strings = {'session_key': session_key, 'name': name}
encoded = urllib.parse.urlencode(query_strings)
return redirect(f"{FRONTEND_URL}?{encoded}")

```

Briefly summarized, the view retrieves the code from the `query_params` dictionary, which is an attribute of the incoming request. If the code was found, an exception is raised. This is an unexpected error that might occur if a user manually visits this URL, or if the identity provider is malfunctioning.

If a code is found, the access token for this user is retrieved. This is done by the `get_access_token` utility, which will be explained at the end of this subchapter. The access token is then used to retrieve the user's personal information using the `get_user_data` utility. Specifically, the data that is retrieved is the data that is added to the application's scope using the Dataporten dashboard, as shown in the *Dataporten* chapter.

The access token and the user's data are stored in the user's session. Sessions are handled automatically by Django and are stored in the postgres database that was installed on the server.

There is a possibility — perhaps a bug — that causes the session to not be saved when written to (as it should be automatically when setting dictionary keys and values), which means that the Session object has no primary key, meaning it does not exist in the database. If so, we call its `save` method to automatically save it to the database, giving it a primary key.

The first name of the user is extracted using the `get_first_name` utility (to be explained). This value will be used on the frontend as a greeting to the user. Because we have had issues with Cross-Origin Resource Sharing (CORS) when sending requests using Ajax in certain browsers, the user's session cannot be found automatically during the POST request. Therefore, we must also send the session key as a query string to the frontend, so that it can be sent back to the backend. That way, the backend can find the user's session key at a later time.

These query strings are URL encoded and appended to the frontend URL (which is locally defined on each server in the `_locals.py` file), and the user is redirected.

The class inherits from Django's built-in `APIView` class. This authentication scheme should not be presumed used in every solution, and the specifics of the implementation may vary, so we have not written an abstract class for this purpose.

ConnectView

The `ConnectView` is the view that handles requests from the frontend. It inherits from the abstract `lotConnectView` class. You will see that all logic contained in this view is specific to the Feide+HiveManager implementation.

Keep in mind that we expect the `get` function is not the handler for PSK requests. The frontend redirects the user to this view (thus resulting in a GET request) if they are found to not be authenticated yet. Also, if a user should happen to visit the `/connect/` URL using their browser, we want to redirect them to Feide if they are not authenticated, or the frontend if they are. That is the purpose of this function. The handler that is used when a user requests a new PSK (which is a POST request, and not a GET request) is defined in the base class, `lotConnectView`, which has already been explained.

The class consists of the following:

```
class ConnectView(IotConnectView):
    authenticator = FeideAuthenticator()
    ad_hoc_adapter = HiveManagerAdapter()
    # Default is True. Added for clarity.
    requires_authentication = True
    permission_classes = []

    def get(self, request, **kwargs):
        session = request.session
        access_token = session.get('access_token', None)
        authenticated = False

        if access_token:
            authenticated = self.authenticator.is_authorized(
                {'access_token': access_token}
            )
        if authenticated:
            user_data = session.get('user_data', [])
            query_strings = {'session_key': session._session_key,
                            'name': get_first_name(user_data)}
            encoded = urllib.parse.urlencode(query_strings)
            url = f"{FRONTEND_URL}?{encoded}"
        else:
            query_strings = {'client_id': DATAPORTEN_KEY,
                            'response_type': 'code'}
            encoded = urllib.parse.urlencode(query_strings)
            url = f"https://auth.dataporten.no/oauth/
                authorization?{encoded}"

        return redirect(to=url)
```

The first step when inheriting from `lotConnectView` is to define which class should be used as the authenticator, and which should be used for the ad hoc adapter. Recall that an exception will be thrown if these attributes are missing. In addition, we choose to require authentication, because we do not wish to let the user generate a PSK for the wireless network if they cannot authenticate themselves using Feide. Besides requiring authentication, we do not wish to impose any permission requirements, and so set the `permission_classes` attribute to an empty list.

The `get` function first attempts to get the access token from the user's session. This will be found if the user is already authenticated. If the access token was found, the value of `authenticated`, which defaults to `False`, will be determined by the authenticator's `is_authorized` function. The access token is used as the authentication data.

If the user is found to be authenticated, the user data is gotten from the session as well, and the user is redirected to the frontend (in the same way as they are in the `DataportenRedirectView` class).

FeideAuthenticator

The `FeideAuthenticator` is the final `Authenticator` class that verifies that the user is authenticated using Feide. If the user is not found to be authenticated, a PSK will not be created.

```
class FeideAuthenticator(Authenticator):
    def validate_data(self, data, **kwargs):
        # Convert to JSON
        data = json.loads(data)
        if self.request.method.upper() != 'GET' and \
            not data.get('session_key', None):
            # session_key is required when posting, because sessions
            # will not work when posting using XmlHttpRequests.
            raise ValidationError({'session_key': 'session_key is
                                   required'})

        return data

    def is_authorized(self, validated_data):
        request = self.request

        # When the method is GET, we do not expect the authentication
        # data to be sent. Handle this case.
        if request.method.upper() == 'GET':
            session = request.session
        else:
            # The method is not GET.
            session_key = validated_data['session_key']
            try:
                # Get the session.
                session = Session.objects.get(
                    pk=session_key
                ).get_decoded()
```

```

        except Session.DoesNotExist:
            # The session key is invalid. Require
                # authentication.
            return False

    # Try to get the access token from the session.
    access_token = session.get('access_token', None)
    if access_token:
        # Try to get the user data from the session.
        session_data = request.session.get('user_data', None)
        # If an access token exists in the session, try to get
        # user data from Feide.
        feide_data = get_user_data(access_token=access_token)
        # Validate
        return self._validate_user_data(session_data, feide_data)

    # access_token is not stored in session.
    return False

    @staticmethod
    def _validate_user_data(session_data, feide_data):
        return session_data and feide_data and \
            feide_data == session_data

```

The `validate_data` function fulfills the `ValidatorMixin`'s method requirements. It first converts the data into JSON. Then, if the request method is not GET, it checks if the data contains a value for `session_key`, which is required due to our problems with CORS when using Ajax in certain browsers. If the session key is not included, an exception is thrown. This function is called by the base class and passed on to the `is_authorized` function.

`is_authorized` is called in `ConnectView`'s `get` function, meaning that the request method may be GET. If so, we can get the session from the request, meaning we do not need the authentication data (which will not be sent with an ordinary GET request). If the request method is not GET, we can expect the session key to exist in the validated data, as is ensured by the `validate_data` method. We use this key to get the session from the database instead.

Once we have the session, we get the user data from Feide. If the data stored in the session differs from that returned by Feide, as validated by the static `_validate_user_data` function, the user is not deemed to be the same person, and is therefore not authorized to generate PSKs using this identity. This is not an expected case, but may occur if the user data has changed. In that case, we require the updated user data. If the two sets of data are equal, the user is authorized.

HiveManagerAdapter

The `HiveManagerAdapter` class is slightly more lengthy, so we will go through its methods one by one. These are its methods and class attributes:

```
class HiveManagerAdapter(AdHocAdapter):
```

```

group_id = 339207927204382

def validate_data(self, data, **kwargs): ...

def perform_generation(self, validated_data): ...

@staticmethod
def _get_response_data(response): ...

@staticmethod
def _get_generation_data(feide_username: str, group_id: int,
                        full_name: str, organization_name: str,
                        policy: str, device_type: str,
                        deliver_by_email: bool, email: str): ...

```

The `group_id` attribute is required by HiveManager's API.

The first function, `validate_data`, fulfills the `ValidatorMixin`'s method requirements. It converts the data into JSON and ensures that a series of required key/value pairs are present in the data. The required keys are `deliver_by_email` and `device_type`. After making these assertions (which result in a `ValidationError` if they are false), the value for the `deliver_by_email` key is converted into a Boolean value (True or False) if it is sent as a string, for good measure.

```

def validate_data(self, data, **kwargs):
    data = json.loads(data)
    if data.get('deliver_by_email', None) is None:
        raise ValidationError("deliver_by_email is required")
    if not data.get('device_type', None):
        raise ValidationError("device_type is required")

    # Formatting
    if isinstance(data['deliver_by_email'], str):
        data['deliver_by_email'] = data['deliver_by_email'].upper() \
            == 'TRUE'

    return data

```

The `perform_generation` method overrides the base method. This is the function responsible for generating the PSK and returning a response optionally containing it. The `url` variable is the fixed URL for HiveManager's API endpoint responsible for creating PSKs. The `OWNER_ID` variable is defined in `_secrets.py`, which is locally defined but synchronized across servers, so that our secrets are not revealed on GitHub. The `authentication_data` variable can surely be found in the request data, as this has been validated in `lotConnectView`'s `dispatch` method. We can also be sure that the `authentication_data` is a dictionary containing a `session_key` key, as was ensured by the authenticator's validation. We get the session from the database instead of the request for reasons already explained relating to CORS.

The `kwargs` variable is declared as a dictionary with many keys and values required by HiveManager's API. When that variable is passed as an argument to a function accepting keyword arguments, the dictionary's key/value pairs are unpacked as if they had been provided thusly:

```
do_something(keyword1=some_value, keyword2=some_other_value)
```

These values are given to the `_get_generation_data` function, which will be explained shortly, and data that is formatted in the way that HiveManager's API expects it is returned. This data is sent to the API endpoint using secret authentication headers (also defined in `_secrets.py`). The response from HiveManager's API is formatted in a way that the frontend expects and returned as a response.

```
def perform_generation(self, validated_data):
    url = "https://cloud-ie.aerohive.com/xapi/v1/identity/
          credentials"
    params = {'ownerId': OWNER_ID}
    auth_data = self.request.data['authentication_data']
    session_key = json.loads(auth_data)['session_key']
    session = Session.objects.get(pk=session_key).get_decoded()

    kwargs = {
        'feide_username': session['user_data']['userid_sec'][0],
        'group_id': self.group_id,
        'full_name': session['user_data']['name'],
        'organization_name': 'Uninett',
        'policy': 'PERSONAL',
        'device_type': validated_data['device_type'],
        'deliver_by_email': validated_data['deliver_by_email'],
        'email': session['user_data']['email']
    }
    data = self._get_generation_data(**kwargs)

    response = requests.post(url=url, params=params,
                             data=json.dumps(data), headers=HEADERS)
    data, status_code = self._get_response_data(response)

    return Response(data=data, status=status_code)
```

The `_get_generation_data` method accepts several keyword arguments. The first thing it does is extract the Feide username from the username found in the user data in the session. The session data's username takes the following form: `feide:username@institution.tld`. We are interested in the `username` part, as there is a maximum length restriction in HiveManager that prevents us from using the complete string.

Most of the keys that HiveManager requires of our request are simply set. Some of the key names are misleading, however, because we are storing different information in these fields than is intended by HiveManager. For our purposes, we are not interested in the separation of first and last name, so we store both in the `firstName` key. We have a free `lastName` field and store the Feide username here for easy searching in HiveManager. HiveManager also

requires that usernames are unique, while we want one user to be able to connect any number of devices. Therefore, we add the current date and time (down to the second) behind the username. In this way we are able to search for users based on their Feide username (which is stored in the *lastName* field) while still allowing a single user to connect multiple devices. We hope that Aerohive will revise these restrictions and offer more freedom.

```
@staticmethod
def _get_generation_data(feide_username: str, group_id, full_name,
                        organization_name, policy, device_type,
                        deliver_by_email, email):
    feide_username = feide_username.strip('feide:').split('@')[0]
    now = datetime.datetime.now()
    hive_user_name = f"{feide_username}:
        {now.replace(microsecond=0).strftime('%y%m%d%H%M%S})"

    return {
        "deliverMethod": "NO_DELIVERY" if not deliver_by_email else
            "EMAIL",
        "firstName": f"{full_name}",
        "groupId": group_id,
        "lastName": feide_username,
        "email": email,
        "organization": organization_name,
        "policy": policy,
        "userName": hive_user_name,
        "purpose": device_type
    }
```

Finally, the `_get_response_data` function checks the status code of the response returned from HiveManager for equality to *HTTP 403 FORBIDDEN*. If so, it probably means that the username is already taken. We ensured that the username would be unique if one second has passed since the last request. This could be an indication that the user has accidentally sent multiple requests somehow, or that there is an attempt to mass produce PSKs. A more thorough way to deal with this is recommended for future developments.

If the status code is not *403 FORBIDDEN*, we assume that the status is *200 OK*. Based on this assumption, we convert the response data to JSON and extract the password, which is returned by HiveManager upon success. If this extraction fails, it means our assumption was wrong. We cannot blame the user if this is a case of a bad request, because it means our validation was not thorough enough. Therefore, we return a *500 INTERNAL SERVER ERROR* response. If everything went according to plan, however, the data will be the generated PSK, and the status code will be *201 CREATED*, which is the standard response when creating objects using POST.

```
@staticmethod
def _get_response_data(response):
    if response.status_code == status.HTTP_403_FORBIDDEN:
        data = {'error': 'Could not generate PSK because the user has
            already created one this second'}
        return data, response.status_code
```

```

try:
    data = json.loads(response.content)['data']['password']
    status_code = status.HTTP_201_CREATED
except TypeError:
    data = {'error': 'Could not generate PSK due to an unknown
                error with the multi-PSK service'}
    status_code = status.HTTP_500_INTERNAL_SERVER_ERROR

return data, status_code

```

Utilities

The last pieces of backend code are the utilities, which are located in the *utils* module of the connect app. There are three utilities (helper functions) here: `get_access_token`, `get_user_data`, and `get_first_name`.

The first function, `get_access_token`, looks like this:

```

def get_access_token(code):
    url = 'https://auth.dataporten.no/oauth/token'
    payload = {'grant_type': 'authorization_code',
               'code': code,
               'client_id': DATAPORTEN_KEY,
               'client_secret': DATAPORTEN_SECRET,
               'redirect_uri': f"{BACKEND_URL}/connect/
                               complete/dataporten/"}

    response = requests.post(url=url, data=payload,
                             headers=DATAPORTEN_HEADERS)
    access_token = json.loads(response.content)['access_token']

    return access_token

```

The goal of the function is to retrieve an access code from Dataporten, which can be used to retrieve user data. It accepts a code, which is sent along the redirect by Dataporten after successful authentication. The URL for retrieving OAuth access tokens from Dataporten is declared at the top.

The function constructs a payload to be sent with a POST request. In this payload, we specify that we wish to use the *authorization code* scheme. There are alternatives, but we found this scheme the simplest. We supply the code, as well as the following local settings: The application's Dataporten client ID, the application's secret key, and the redirect URI, which is already defined in the Dataporten dashboard. The redirect URI must match the previously given redirect URI in order for the request to be accepted.

The payload is sent to Dataporten, and the response is gotten. We retrieve the access token from this response by converting it to a JSON dictionary and using square brackets to retrieve the value of the *access_token* key. This key is returned.

The second function, `get_user_data`, accepts an access token. Its goal is to retrieve the user's data using the access token, which is stored in the user's session. Dataporten's endpoint URL for retrieving user data is declared at the top:

```
def get_user_data(access_token):
    url = 'https://auth.dataporten.no/userinfo'
    headers = {'Authorization': f'Bearer {access_token}'}

    response = requests.get(url=url, headers=headers)
    content = json.loads(response.content)
    return content.get('user', None)
```

We construct an authorization header that uses the access token as a *bearer* token. We send a GET request to the Dataporten endpoint with this header, and the user's data is returned. We construct a JSON dictionary of the response and retrieve the value of the *user* key.

The final utility, `get_first_name`, accepts user data and returns a best guess for the user's first name:

```
def get_first_name(user_data):
    name = user_data.get('name', 'bruker')
    return name.split(' ')[0]
```

This is done by first extracting the value of the *name* key from the user data. *name* is one of the scopes we added to our Dataporten application. The name is then split at every occurrence of a space, turning it into an array of words. We return the first word, as we assume that the first word in a full name is the first name. There are likely better solutions that should be considered in the future.

Frontend components

The frontend is an important part of our solution. While it can be customized in any desirable way, there must be a frontend of some sort. We chose a simple form with a *submit* button. Because we do not want the user to experience redirects after filling in the form and clicking the button, we send an asynchronous request using JavaScript. We let the user know that the request is being sent by showing a “spinning wheel” before the newly generated PSK is displayed, or any eventual error message.

There are two main components of the frontend: 1) The markup and styles, and 2) the code sending requests to the backend, validating data, showing and hiding content, etc.

We will not go into great detail regarding markup and styling, as these are independent of the core technology. We chose this design in the design stage because of its simplicity, and because we made a conscious decision to not be overly ambitious in the research stage.

Markup and styles

The markup (in our case written in HTML, which stands for HyperText Markup Language) of a page gives instructions about the page’s contents and structure, while the style (in our case, CSS) gives instructions regarding the appearance of those elements. Both are usually necessary for a complete page.

The frontend is available at <https://github.com/Uninett/IoTConnect-FrontEnd>.

Brief description of HTML

The HTML document, *index.html*, includes, among other things:

- The “main form”: A container with an input for a description of the device and a check box for specifying whether or not the PSK should be sent by email,
- a “loader” that is animated using CSS (hidden initially), and
- an “alert box” that is hidden initially.

CSS

Cascading style sheets (CSS) is the styling language that is used almost everywhere on the web. It supports inheritance, meaning that an element may inherit some styles from its parent element while having some unique styles of its own.

Styles are usually stored in files with the *.css* extension and linked in the `<head>` tag of the HTML document thusly:

```
<link rel="stylesheet" href="[path to stylesheet]">
```

In the HTML document, we assign classes and IDs to elements. Classes, IDs and tags can all be styled using CSS.

We have linked two style sheets. The first, *styles.css*, is our own style sheet at 172 lines. The second, *styles_check.css*, is a collection of styles found at W3Schools that enables us to draw custom styles for checkboxes.¹²

We will not explain the structure or contents of our style sheets. Suffice it to say that we have attempted to make the style sheet as brief as possible through the utilization of inheritance. The complete style sheets can be found in our frontend's Git repository, which is hosted on GitHub.¹³

Responsive content

The frontend's HTML elements are relatively large. Some displays, especially mobile screens, are too small to display the full 845 pixel width of the form container. Therefore, we have added the following CSS property to the style of the form container:

```
max-width: 100%;
```

This ensures that the container may only cover 100% of the full width of the page.

Additionally, we must decrease the font size of the welcome header as the display's width decreases. This is done using media queries, which are new in CSS3. Media queries allow us to use different styles for elements based on the size of the display, for example.

These styles dictate the font size of the welcome header, which as an element of type `<h1>`:

```
h1 {
  ...
  font-size: 60px;
  ...
}

@media only screen and (max-width: 844px) {
  h1 {
    font-size: 45px;
  }
}

@media only screen and (max-width: 600px) {
  h1 {
    font-size: 35px;
  }
}
```

¹² The styles were found at https://www.w3schools.com/howto/howto_css_custom_checkbox.asp. Retrieved on April 29, 2019.

¹³ Available at <https://github.com/Uninett/IoTConnect-FrontEnd>.

The first style provides a default font size for the header. The second style takes effect only when the width of the display is 844 pixels or less, and reduces the font size from 60 pixels to 45 pixels. The final style takes effect when the width of the display is 600 pixels or less, and reduces the font size further to only 35 pixels. The result on a small screen is that the container shrinks to fit 100% of the viewport if the viewport's width is less than 845 pixels, which is the width of the container, instead of extending beyond the edge of the viewport:



Please note that the unit *em* is preferably used for text. An argument for using pixels instead is that it gives us greater control of the final look of the page. An argument for using the *em* until is that it makes it easier for the visually impaired to scale text up or down. This is a header with a much larger than normal font size. Therefore, we do not anticipate a need to scale it up further. The *em* unit can also be used to achieve something similar to media queries, but with a gradual scaling.

Accessibility

All websites should be accessible to the visually impaired. Screen readers — assistive technology (AT) used by the visually impaired to provide a speech-based representation of websites and navigation — make use of HTML tags and properties that are often forgotten by those who do not need them. One of these properties is the *lang* property of the `<html>` tag, which declares the language of the page. In our case, this is Norwegian, so our `<html>` tag looks like this:

```
<html lang="no">
```

Furthermore, screen readers work best when the page's HTML is semantically written. When code or markup is semantic, it means that its components and structure are easily recognized by looking at the names, types and order of elements. For example, these two elements achieve roughly the same effect:

Example 1:

```
<h1 style="font-size: 60pt;">Welcome, user</h1>
```

Example 2:

```
<div style="font-size: 60pt;">Welcome, user</div>
```

The main difference is that the latter is not semantic. There is no expectation that a `<div>` element acts as a header, while the `<h1>` tag is designed for the most important headers. `<h2>` is designed for sub-headers of the `<h1>` chapter, and so on. Therefore, we used an `<h1>` element for our header.

Likewise, our submit button could easily have been a `<div>` element. Instead, we use the `<button>` element, as it is recognized as a clickable button by screen readers. The *toggle* "button" (*show/hide password*) is an `<a>` element, which is also expected to be clickable, as it is used for links. The same principle is applied to all inputs and elements that the user can interact with. It should be mentioned that the visually impaired should not have to click the *show/hide password* button, as this simply changes how the string within a text field is displayed, and not the actual value of the string, which is readily available to screen readers inspecting the HTML source.

JavaScript

JavaScript is a Turing complete programming language for client-side scripting (usually) in the browser. It allows us to programmatically alter the contents of the page, perform arbitrary computations, send requests asynchronously, and more. We have used JavaScript for these purposes, including some validation, asynchronous requests and response handling, and hiding, showing and altering page elements and their contents.

Please note that, in most cases, the elements of the HTML are available as global variables due to these lines:

```
var descriptionInput = document.getElementById("description");
var deliverByEmailInput = document.getElementById("send-email");
var submitButton = document.getElementById("sendBtn");
var pskTextField = document.getElementById("psk");
var showPskButton = document.getElementById("showPsk");
var form = document.getElementById("outerForm");
var loader = document.getElementById("loader");
var pskScreen = document.getElementById("pskScreen");
```

This is an optimization that was put in place so that engine does not need to look through the document repeatedly for the same element. We know that the ID of elements will never

change in our document. An element would normally be retrieved using `document.getElementById`, but we will simply refer to these variables.

Initial

When the page is loaded, a script — *initial.js* — is run. The script is placed at the bottom of the body, ensuring that the HTML elements are declared before it is executed.

```
// Get username from URL parameters
var username = getUrlVars()['name'];

if (username == null) {
    // HTTP redirect
    window.location.href = backendUrl + "/connect/";
};

// Set welcome text
var welcomeHeader = document.getElementById("welcomeText");
welcomeHeader.innerHTML = "Velkommen, " +
decodeURIComponent(username);

// Add event listeners to input elements
var inputs = document.getElementsByTagName("input");
for (i = 0; i < inputs.length; i++) {
    inputs[i].addEventListener("keyup", function(event) {
        if (event.keyCode === 13) {
            event.preventDefault();
            document.getElementById("sendBtn").click();
        }
    });
}
```

First, the script gets the *name* query string from the URL. If it does not exist, the user is redirected to the backend for authentication.¹⁴ If the name is found (as it will be if the user was redirected by the backend), the welcome header's text is changed to "Welcome, [name]".

Next, the script finds all elements declared with the `<input>` tag. Such elements are used for text fields, buttons, drop-down lists, and other typical input elements that the user can interact with without the use of custom code. The script then iterates through the list of input elements and adds an event listener to each one. An event listener is a function that is called when an event occurs. Here, we are adding an event listener to the *keyup* event, which happens when a key has been pressed and becomes unpressed. We assign a lambda function (a function that is not immediately executed, but passed as an argument) as the

¹⁴ We would prefer that the user is presented with a white screen before redirection. Instead, they see the frontend with the following message: "Welcome, undefined". We have concluded that we do not have time to make this change.

event listener. JavaScript allows us to declare lambda functions directly in this manner, meaning that we do not need to declare it elsewhere and provide its name. We are declaring a nameless lambda function.

Once triggered, the event listener will check the event's `keyCode` property, which we expect the `keyup` event object to have. If the `keyCode` property of the event is 13, it means the key in question is the *enter* key.

In other words, this code is triggered when the user has focused on one of the inputs on the page and pressed the *enter* key. The user would expect the action of pressing the *enter* key to equate to clicking the *submit* button (whose ID is "sendBtn"), so we click it programmatically.

We decided to use this loop because we initially had an additional input field; we also had an email field. If more input fields are added in the future, we do not have to explicitly add this event handler to the `keyup` event.

Additionally (and independently of the above), we expect a query string named *name* to be present in the URL. This query string is added in the redirect by the backend. The `welcomeHeader` element is an `<h1>` (primary header) element that says, "Welcome, [name]". The user's name must be extracted from the URL, and the `<h1>` element's inner HTML must be replaced with a new welcome message including that name.

We included a function that extracts the query strings from the URL in the links in the `<head>` element of the HTML document, meaning it is available at this point:

```
function getUrlVars()
{
    var vars = [], hash;
    var hashes = window.location.href.slice(
        window.location.href.indexOf('?') + 1
    ).split('&');
    for(var i = 0; i < hashes.length; i++)
    {
        hash = hashes[i].split('=');
        vars.push(hash[0]);
        vars[hash[0]] = hash[1];
    }
    return vars;
}
```

We found this script at

<http://jquery-howto.blogspot.com/2009/09/get-url-parameters-values-with-jquery.html>. It was written by Snipplr user Roshambo (<https://snipplr.com/users/Roshambo/>).

It returns a dictionary containing all (in our case two) query parameter names as keys and their corresponding values. We extract the value for the *name* key from this dictionary using

square brackets before changing the welcome header's inner HTML (the HTML or plain text that appears between the opening and closing of the element).

Validation

Before the user is allowed to send the form data to the backend for processing, we validate their input data using JavaScript and HTML5. This saves the user time and prevents frustration, and it spares our servers from requests that are bound to result in failure. In our case, there is only one field that may contain invalid input, namely the *description* field. We require that the description is at least one character long, and at most 50 characters long.

The first line of defence against incorrect form data is HTML5's validation functionality. We placed the 50 character limit in the HTML:

```
<input type="text" class="formInput" id="description"
name="description" onkeyup="setDescriptionColors()" maxlength="50">
```

There is no native *minlength* attribute. It is possible to use a regex pattern for this purpose, but we do not want to check for validity of input data before the user has changed the value of the input field or clicked the submit button, because we do not want the field to appear incorrect before the user has been given a chance to enter a value. We perform this validation in JavaScript.

Notice that we have assigned a handler to the *onkeyup* event. This handler is the `setDescriptionColors` function:

```
function setDescriptionColors() {
  if (!descriptionIsValid()) {
    descriptionInput.style.border = "1px solid red";
    submitButton.disabled = true;
  }
  else {
    descriptionInput.style.border= "1px solid white";
    submitButton.disabled = false;
  }
}
```

This function calls the following function to test the validity of the description field's input data:

```
function descriptionIsValid() {
  // Check if the inputted description is valid
  return (descriptionInput.value.length >= 1);
}
```

The first function sets changes input element's border color to red and disables the submit button if the input is invalid. If the input is valid, these changes are reverted. The second

function simply returns true if the description is at least one character long, and false otherwise.

We use *onkeyup* instead of *onkeydown* here because handling *onkeydown* would produce a red border (which is how we signify invalid input data) when a key is first pressed, even though the value would be valid. This is because the character is only added to the input element's value *after* the key has been pressed. Instead, the field will be given a red border if the user erases their initial input, and the user will be unable to click the submit button.

POST request

When the user has entered a valid description for their device and clicked the submit button or pressed the *enter* key, a request is sent to the backend using this code:

```
function sendData() {
    loading();
    // Get query strings
    var url_vars = getUrlVars();

    // Authentication data
    var session_key = url_vars['session_key'];

    // Generation options
    var description = descriptionInput.value;
    var deliver_by_email = deliverByEmailInput.checked;

    // Request data
    var authentication_data = {
        "session_key" : session_key
    };
    var generation_options = {
        "device_type" : description,
        "deliver_by_email" : deliver_by_email,
    };
    var data = {
        authentication_data: JSON.stringify(authentication_data),
        generation_options: JSON.stringify(generation_options)
    };

    // Create request
    var xhr = new XMLHttpRequest();
    xhr.open('POST', backendUrl + "/connect/", true);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.withCredentials = true;

    // Handle request responses
    if (xhr.status === 201) {
        // Created (success)
        showPskScreen(xhr.responseText);
    }
}
```



```

else if (xhr.status == 401) {
    // Unauthorized
    window.location.href = backendUrl + "/connect/";
}
else if (xhr.status == 500) {
    // Internal server error
    showAlert('Det har oppstått en uventet feil. Vennligst prøv
              igjen.');
```

showFormScreen();

```

}
else if (xhr.status == 400) {
    // Bad request
    showAlert('Noe er galt. Vennligst bekreft at e-postadressen er
              riktig.');
```

showFormScreen();

```

}
else if (xhr.status == 403) {
    // Forbidden (too fast)
    showAlert('Du etterspør passord for hyppig. Vennligst prøv igjen
              senere.');
```

showFormScreen();

```

}
else if (xhr.status == 408 || xhr.status == 404 || xhr.status == 503) {
    // Timed out, not found, or unavailable (HiveManager or Dataporten
    // not responding)
    showAlert('Tjenesten er midlertidig utilgjengelig.');
```

showFormScreen();

```

}

// Handle request error
xhr.onerror = function() {
    alert('Unexpected error. Please check your browser.');
```

};

```

// Send the request
xhr.send(JSON.stringify(data));
}

```

The function constructs an `XmlHttpRequest` containing the required authentication data and generation options, as retrieved from the form and query strings. When declaring the `XmlHttpRequest` object, we also pass a function that handles the response. The code that handles the various response status codes is not executed before the request is fully processed.

We also add an error handler, which occurs when non-standard errors occur, such as the browser refusing to send the request.

Most handlers, as you see, display a message to the user and shows the initial form (so that the “spinning wheel” is not spinning forever). The handler for the 201 (CREATED) status code, however, calls `showPskScreen`, which will be described shortly. The handler for the

401 (UNAUTHORIZED) status code redirects the user to the `/connect/` endpoint, which is expected to redirect the user to Dataporten's single sign-on page for authentication.

The conditions are ordered by how likely we believe they will occur, placing the most likely first. This minimizes the average number of conditions that need to be checked before the appropriate condition is found.

Visual feedback

The `sendData` function initially calls the `loading` function before sending the data and processing the response. The `loading` function is responsible for replacing the current content on the page with a *spinning wheel*, indicating that a process is running that must finish. This is the function:

```
function loading() {
    form.style.display = "none";
    loader.style.display = "block"
    pskScreen.style.display = "none";
}
```

`form` is the element containing the input elements. `loader` is an element that is animated using CSS transitions (representing the “spinning wheel”). The form is hidden along with the “PSK screen” (which is expected to eventually display the returned PSK), and the `loader` element is displayed by changing its `display` property from its initial “none” state to the default “block” state.

If the response fails, or the response's status code is not 201 (CREATED), an alert may be displayed by calling the `showAlert` function:

```
function showAlert(msg) {
    alertMsg.innerHTML = msg;
    alertBox.style.display = "block";
    nbAlerts++;
    setTimeout(hideAlert, 15000);
}
```

The function accepts a message to be displayed. There is a label element whose ID is `alertMsg`. The function replaces this label's inner text with the message. The label exists within a container whose ID is `alertBox`. This container is initially hidden. The function displays it by changing its `display` property to “block” instead of the initial “none”.

Next, it increments a global integer, `nbAlerts`. Its purpose will become apparent promptly. Finally, it creates a timer with a duration of 15 seconds, after which the `hideAlert` function is called:

```
function hideAlert() {
    nbAlerts--;
    // Only hide if this is the last alert
```

```

    if (nbAlerts == 0) {
        alertBox.style.display = "none";
    }
}

```

This function decrements the counter, after which its value may be zero or positive. If the value is positive, it means that a second alert has been triggered before the 15 seconds have elapsed. In that case, we should not hide the element, because each alert should be displayed for 15 seconds before being hidden. Disregarding this would result in the second alert being displayed for less than 15 seconds.

If the value is zero, it means that this is the last alert triggered (15 seconds ago), and the element is hidden.

An alert looks like this:



It can be immediately closed by clicking the X icon.

We followed a tutorial by W3Schools in order to create this alert system. The tutorial can be found at https://www.w3schools.com/howto/howto_js_alert.asp.

Displaying of PSK

When the backend has given its response to the request, several status codes are possible, as is shown in the *POST request* sub-chapter. If the status code is 201 (CREATED), a function named *showPskScreen* is called, and the response text is passed as an argument. This response text is the PSK. This is the *showPskScreen*:

```

function showPskScreen(password) {
    form.style.display = "none";
    loader.style.display = "none"
    pskScreen.style.display = "block";
    pskTextField.value = password.replace(/['"]+/g, '');
    // Clear the password after 5 minutes
    setTimeout(clearPsk, 300000);
}

```

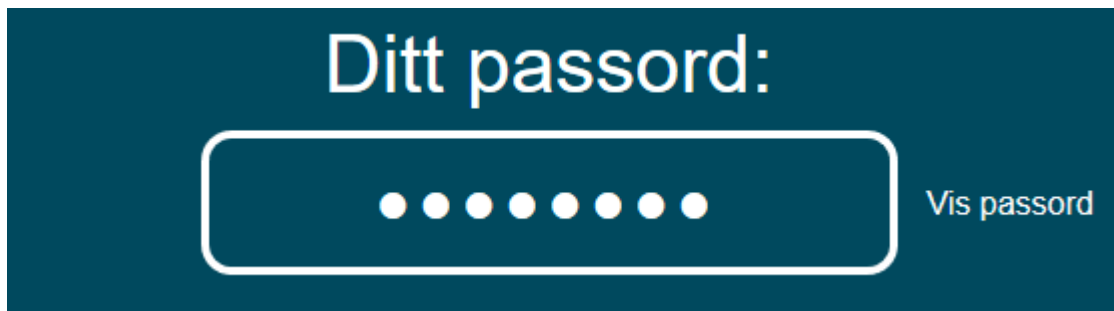
First, it hides the *form* element (which contains the input fields) by settings its *display* property to "none". This should also hide it from screen readers. It does the same to the *loader* element, which is currently spinning. The class *pskScreen* is assigned to an element

whose *display* property is currently “none”. We give it the default *block* display instead, showing it to the user.

Next, the element whose class is *pskTextField*, which is a text field that has been disabled (meaning the user cannot change its text), is given the value of the PSK. Giving a text field a value means changing the text within it. Before the PSK is assigned to its value, we remove any quotation marks surrounding. Due to how the XMLHttpRequest class expects returned data, this must be done. The response is returned as JSON, while plain text is expected. This causes the quotation marks of the JSON to be present in the string. We found that this is the simplest way of dealing with this given limited time.

Finally, we create a timer using `setTimeout`. The timer will run for 300000 milliseconds, which is equal to five minutes. Once this time has elapsed, the `clearPsk` function will be called. The `clearPsk` function simply replaced the value of the *pskTextField* element to “expired”. This is done so that the password is not visible to unauthorized persons if the computer is left unattended.

This is the result:

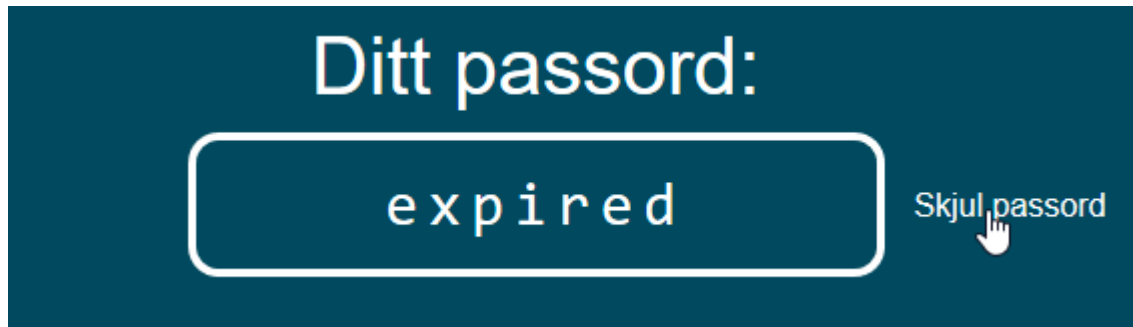


The *show password* button is an `<a>` element whose *onclick* event is handled by the function `togglePsk`:

```
function togglePsk() {
  // Show/hide the password
  if (pskTextField.type === "password") {
    pskTextField.type = "text";
    showPskButton.innerHTML = "Skjul passord";
  } else {
    pskTextField.type = "password";
    showPskButton.innerHTML = "Vis passord";
  }
}
```

There are two possible cases: 1) The element’s type is “password”, or 2) the element’s type is *not* “password”. In the former case, the type is set to be “text”, and the inner HTML of the `<a>` element is set to “Hide password” (in Norwegian). In the latter case, the type is set to be “password”, and the inner HTML of the `<a>` element is set to “Show password”.

When the type of an input field is "password", the characters are automatically displayed as dots, as seen in the screenshot on the previous page. Therefore, the effect of clicking the "Show password" button is that the characters are revealed or concealed by changing the type of the element. This is the revealed state after five minutes, at which point the PSK is replaced for safety:



Testing and verification

Testing has been performed throughout the project. Once we were near finalization, we performed more thorough testing and found a handful of issues. Testing consisted of unit tests and manual testing procedures.

Unit tests

Unit tests are procedures that are executed with an expectation of results. In other words, a unit test performs some actions and makes assertions about the result of those actions. If any assertion is incorrect, an exception is raised and the test has failed.

Our unit tests mainly assert the responses of our endpoints. A response is gotten by sending a request to these views. These requests are not sent through a network, but constructed programmatically, allowing us to pretend like they are real requests received in a production environment. We will call such tests *API tests*.

Additionally, the *iotconnect* package contains a few tests that test the view classes themselves without sending requests. We will call these tests *class tests*.

First, let us provide an overview of the *lotConnectTestCase* class, which provides a few helpful functions. We have collapsed the functions in this class and will describe some of them in greater detail:

```
class IotConnectTestCase(APITestCase):
    url = ''

    def get(self, query_params=None): ...

    def post(self, data=None, **kwargs): ...

    def patch(self, data=None, **kwargs): ...

    def delete(self, query_params=None): ...

    def assert_status_code(self, response, status_code: status): ...

    def assert_redirect(self, response, redirect_pattern: str = None): ...

    @staticmethod
    def _get_extra_headers(): ...
```

First, let it be said that our unit tests will inherit from this class, meaning that the methods shown above are available to all unit tests. The names of the methods should give a clue as to what the individual method's purpose is.

`get`, `post`, `patch`, and `delete` simulate requests to an endpoint. The `get` method simulates a GET request, for example. The HTTP methods POST and PATCH are typically

accompanied by a body, while GET and DELETE do not. Therefore, `post` and `patch` accept a `data` argument, which will constitute the request's body. Here goes, for example, the authentication data and generation options. A GET request, while not sending any data, may yet affect the outcome of the request by changing the value of query parameters. Query parameters, as has been explained previously, take the form of a keyword and a value in the URL, for example: `example.com?key1=value1&key2=value2`. This can be used for filtering the results of the GET request, for example. Meanwhile, a POST request typically contains data describing an object to be created. This data may take the form of a dictionary containing keys and values. For a user, this data may look something like this:

```
data = {
    'first_name': 'Ola',
    'last_name': 'Nordmann'
}
```

This dictionary is encoded into a string before it is decoded back into a dictionary at the backend for further interpretation.

That should be sufficient to explain the purpose of the `get`, `post`, `patch`, and `delete` methods. We did not find it necessary to add helper functions for other methods, such as OPTIONS, PUT or HEAD. Our reasoning should become apparent soon. First, a look at the `patch` method, which greatly resembles the other three request method helpers:

```
def patch(self, data=None, **kwargs):
    data = json.dumps(data)
    response = self.client.patch(self.url, data,
                                content_type='application/json',
                                **self._get_extra_headers(),
                                **kwargs)

    return response
```

First, it converts the data into a string, as is done before the data is transmitted. It will automatically be decoded by Django upon arrival. Next, it uses the `client` attribute (and instance of a Django built-in class that inherits from `RequestFactory`, another built-in class) of the built-in `APITestCase` class to create a PATCH request headed for the `self.url` attribute (which is set in the unit test, as you will see) with the content type of `application/json` and the header returned by the `_get_extra_headers` function:

```
@staticmethod
def _get_extra_headers():
    return {'HTTP_ACCEPT': f'application/json'}
```

The response returned by the view is finally returned. A unit test may, for example, assert that the method PATCH is not supported by the view.

As mentioned, the passing or failing of unit tests typically depends on the assertions made by the test. At the end of a unit test, it may be asserted that the response's status code is equal to 200 (which signifies the HTTP 200 OK response). Sometimes, assertions are so complex or frequent that a function should exist to aid us in making the assertions with less

code, in the spirit of the Don't Repeat Yourself (DRY) principle. Therefore, we added the `assert_status_code` and `assert_redirect` methods. The former accepts a response and an expected status code as arguments and makes a simple assertion:

```
def assert_status_code(self, response, status_code: status):
    self.assertEqual(status_code, response.status_code)
```

Please note that `assertEqual` is a method of the `APITestCase` class, which `lotConnectTestCase` inherits from. It raises an exception if the assertion is false, i.e the first and second arguments are not equal. Here, we assert that the expected status code and the response's actual status code are equal. If not, an exception is raised, and the test fails.:

The latter, `assert_redirect`, asserts that the response is redirecting us to some URL. Additionally, it lets us make assertions about the URL to which we are redirected:

```
def assert_redirect(self, response, redirect_pattern: str = None):
    self.assertTrue(isinstance(response, HttpResponseRedirect))

    if redirect_pattern:
        self.assertIsNotNone(re.match(redirect_pattern, response.url))
```

First, it asserts that the response is an instance of `HttpResponseRedirect`, a built-in class for redirect responses. Next, if the programmer supplied a pattern, it asserts that the pattern supplied matches the redirect URL.

The `match` function of the `re` interface returns `None` if the supplied pattern does not match the supplied target string. Otherwise, it returns one or more matches, and information about the location of those matches. We are interested in finding out if there is a match between the supplied redirect pattern and the actual redirect URL of the response, and therefore assert that the return value of `match` is not `None`.

API tests

Our API tests are located in the `connect` app — specifically in the `test_api.py` module of the `tests` package of the `connect` app. There are two test classes here: 1) `ConnectViewTest`, and 2) `DataportenRedirectViewTest`. As the names imply, these test classes test the two endpoints included in our custom solution.

Before we list the individual unit tests within these classes, let us describe one test case that asserts the status code of the response, and one test case that asserts the redirect URL of the response:

```
def test_post__nonsensical_session_key__forbidden(self):
    data = {
        'authentication_data': {'session_key': 'some_data'},
        'generation_options': self._get_valid_generation_options()
    }
    response = self.post(data)
    self.assert_status_code(response,
```



```
status_code=status.HTTP_403_FORBIDDEN)
```

Notice the name of the method. Each part is separated by two underscores. The first part states which HTTP method is tested. It is important that the test's name begins with *test*, so that it is recognized as a test by the code editor. The middle part, if any, names the state that should give the expected result. In this case, a nonsensical session key is expected to give the result that is named in the last part of the name: A *HTTP 403 FORBIDDEN* response.

First, it constructs some data to be sent in the request's body. This data consists of authentication data containing a nonsensical session key, as well as some valid generation options.

The response is gotten using the `post` helper method before we assert, using the `assert_status_code` helper method, that the response's status code is 403. This is because nonsensical authentication data will result in unsuccessful authentication.

It should also be mentioned that the URL of the `post` method is known because of this built-in function, which is run before every test:

```
def setUp(self):
    self.url = reverse('connect')
```

You may recall that we named our endpoints when declaring the views. The endpoint handled by `ConnectView` was named *connect*. The `reverse` function finds the URL of a view based on the URL's name.

The `_get_valid_generation_options` function simply returns the following data:

```
{
    'deliver_by_email': True,
    'device_type': 'Refrigerator'
}
```

These are all our API tests:

```
def test_patch__method_not_allowed(self): ...
def test_delete__method_not_allowed(self): ...
def test_get__not_authenticated__redirected_to_dataporten(self): ...
def test_post__no_data__bad_request(self): ...
def test_post__some_data_but_not_required_data__bad_request(self): ...
def test_post__auth_data_but_not_generation_options__bad_request(self): ...
def test_post__nonsensical_authentication_data__bad_request(self): ...
def test_post__nonsensical_session_key__forbidden(self): ...
```

The methods tested, the initial states and the expected result should be apparent from the method names.

Class tests

Unlike the API tests, the class tests make assertions against the classes found in the core solution, and not the custom views. Therefore, they are found in the `test_view_classes.py` module of the `test` package of the `iotconnect` package. They also do not inherit from the `lotConnectTestCase` class, as they do not make use of any of the helper methods and are not API tests.

We do not see the necessity of describing the logic of these tests, as feel that a list of methods will suffice. These are the class tests:

```
def test_class__view_set_up_without_adapter__attribute_error(self): ...

def test_class__view_set_up_without_authenticator_and_authentication_is \
    _required__attribute_error(self):

def test_class__view_set_up_without_authenticator_and_authentication_is \
    _not_required__not_attribute_error(self):
```

These tests all passed and so did not result in any changes.

Resulting changes

Some tests did not pass initially. The failure of these tests resulted in changes to our source code.¹⁵

Briefly summarized, these changes included:

- Adding a custom middleware class that handles the custom `NoDataportenCodeError` exception,
- Raising of this custom exception instead of `ValueError`,
- Changes to exception messages, and
- Raising `ValidationError` if the request data is `None`.

Manual testing procedures

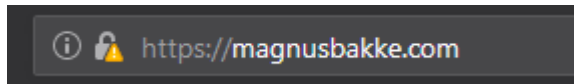
Unit tests should not and often cannot test states that depend on the environment. We want the unit tests committed to our Git repository to succeed regardless of the environment. Yet, there are tests that need to be performed that *do* depend on the environment. For testing such aspects of the solution, we have devised a short series of manual testing procedures.

¹⁵ These changes can be seen in this commit:

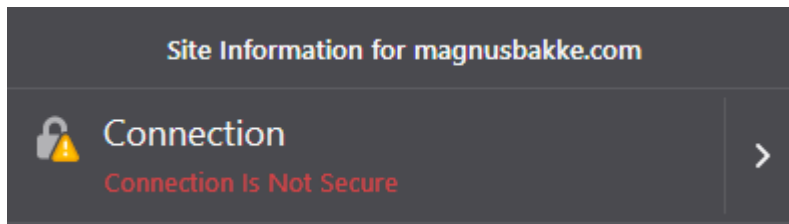
<https://github.com/Uninett/loTConnect-Backend/commit/01882930500e2dff7ab20e360ea46004dd3e939>

Verification of proper SSL encryption

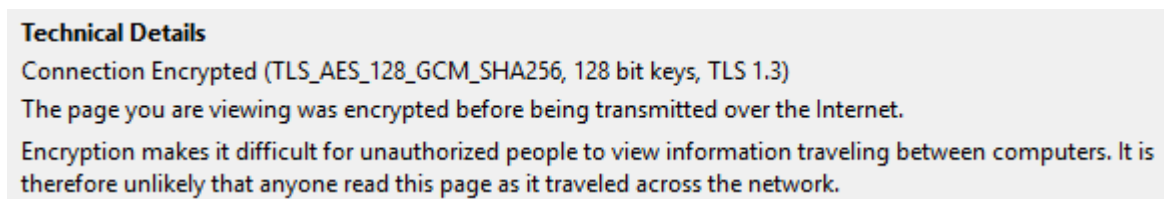
When entering the frontend of the testing environment (using Firefox), this padlock appears:



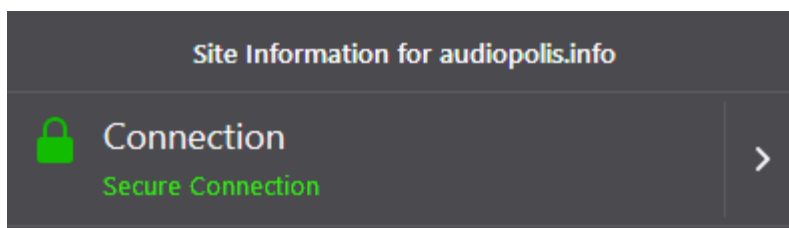
If we click on the padlock, we see the following:



This is expected. This is because the staging environment of Let's Encrypt is not trusted by the browser. If we click the rightwards arrow, then *more information*, we can see that the connection is indeed encrypted:



The production server uses the production environment of Let's Encrypt's API for certificate requests, and therefore does not display the same warning:



These are both expected results; Caddy is built for "automatic HTTPS", which involves supplying a certificate authority, such as Let's Encrypt, and doing minimal configuration. Furthermore, the Django applications on the testing and production servers are configured to enforce HTTPS, which means that the service will not work if a secure connection cannot be established.

Testing of encryption between client and access point

In order to monitor Wi-Fi traffic for the purposes of this test, we need an compatible “WiFi sniffer” interface¹⁶. We are not currently in possession of such an interface and are unable to obtain one in reasonable time and at a reasonable expense. Therefore, we are unable to perform this test. We do, however, most definitely expect different PSKs to result in different hashes, rendering sniffing impossible.

Testing of roaming

In order to verify that we are able to roam between access points, we powered up both access points. A computer was connected to *IoT-roam*. We verified that we had access to the internet. Next, one of the access points were switched off. We verified that we still had access to the internet. Next, the access point that was switched off was switched on. We waited for the access point to boot. Finally, the second access point was switched off. We verified that we still had access to the internet.

Ergo, we are able to roam between access points.

Testing of traceability

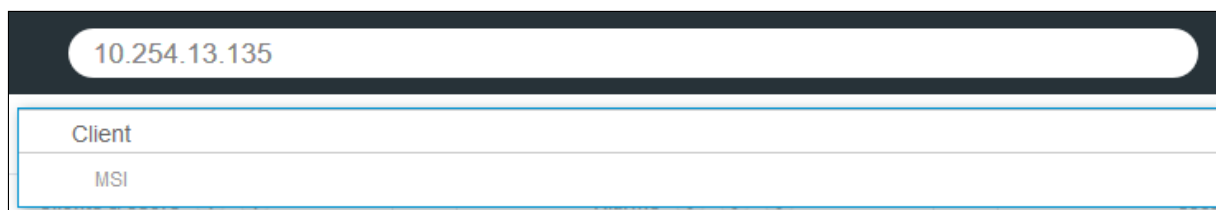
The detection of abuse on a network is a task for network components such as network/host intrusion detection systems (NIDS and HIDS). If an IP address on the network is found to be exerting malicious behaviors, it is crucial that the owner of the IP address can be found.

In HiveManager, there is a magnifying glass icon at the very top of the page:



When clicking the icon, a search field appears.

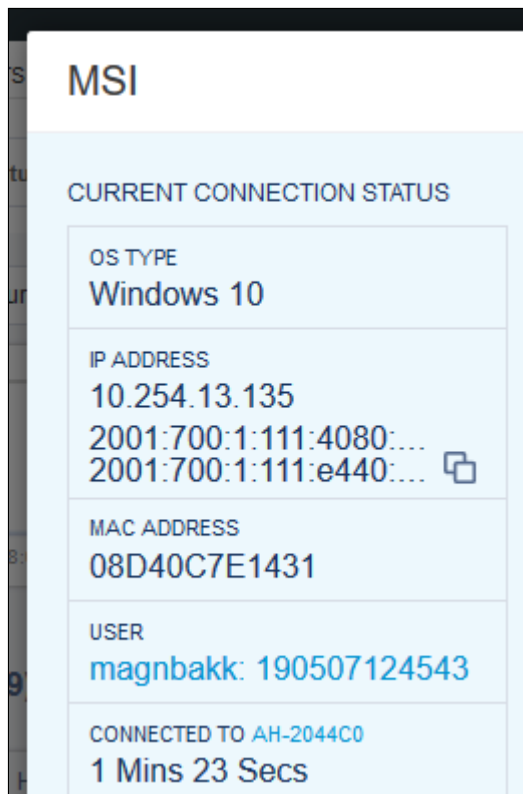
We will pretend like the IP address of Magnus has been found to be exerting malicious behavior, and will attempt to find his identity by searching for the IP address. The matching clients appear automatically underneath the search field:



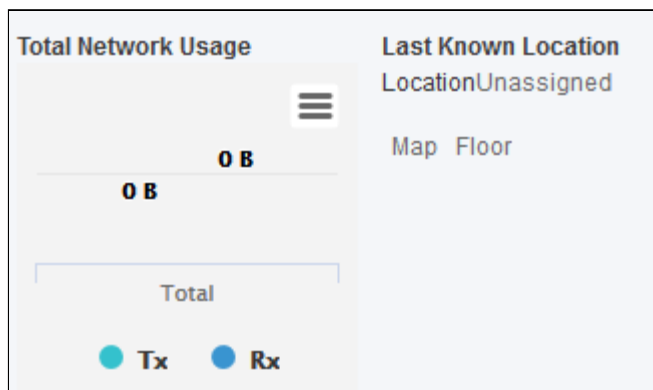
Clicking on the result yields an exhaustive overview. Here is an excerpt:

¹⁶ See this example:

<https://www.amazon.com/OpenPcap-Sniffer-Wireless-Capture-Alternative/dp/B07JCJHHS4>



Furthermore, if we click on the username, we are given an overview that includes details about the user's location and behavior:



This would be useful if we had created a floor plan, which we have not.

This test shows that users are traceable when using IoT-roam.

Integration tests

Integration tests are testing procedures that do not test individual units, but rather multiple units put together. While a unit test may pass, the same functionality that is tested within it might fail in a realistic context. Therefore, we perform integration tests, which involve actually using the product.

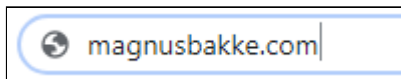
The primary integration test involves these steps:

- 1) Clear cookies and cache
- 2) Visit the testing frontend
- 3) Await redirection to Dataporten
- 4) Enter credentials
- 5) Await redirection to frontend
- 6) Enter a device description
- 7) Choose delivery by email
- 8) Click submit
- 9) Await the PSK on the frontend
- 10) Show the PSK
- 11) Find the email containing the PSK
- 12) Verify that the PSK field reads “expired” after five minutes

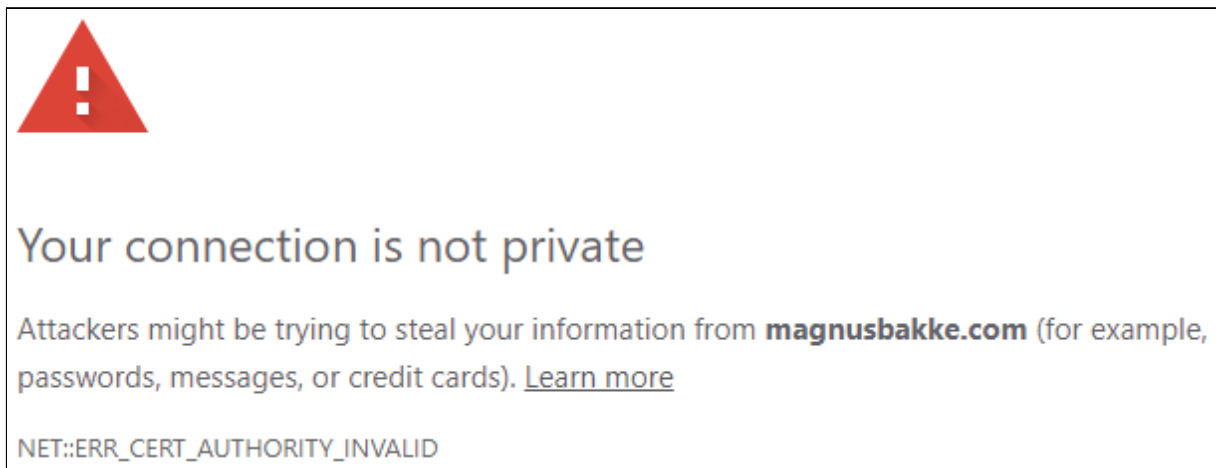
If all these steps succeed, the integration test is considered successful.

Here is a summary of the current result (as of May 3rd, 2019) of this integration test:

After clearing cookies and cache in Google Chrome, we visit the testing server at magnusbakke.com.



We expect a security warning due to the use of Let's Encrypt's staging environment:




We ignore this warning and continue. As expected, we are redirected to Dataporten. We enter our credentials:

Log in with Feide




You need to log in via Feide to access IoTConnect Testing.

Choose your affiliation 

 NTNU 

Remember me

Continue 

We expect to be redirected to the frontend:

Velkommen, Magnus

Kort beskrivelse av enheten

Send passord via e-post

Hent passord

We enter a device description and keep the checkbox checked:

Velkommen, Magnus

Kort beskrivelse av enheten

Refrigerator

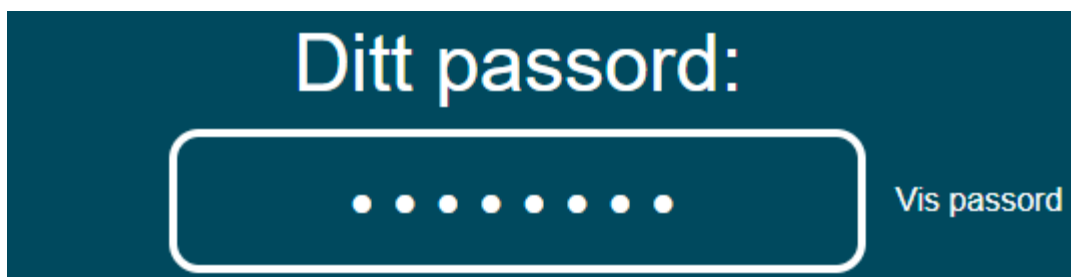
Send passord via e-post

Hent passord

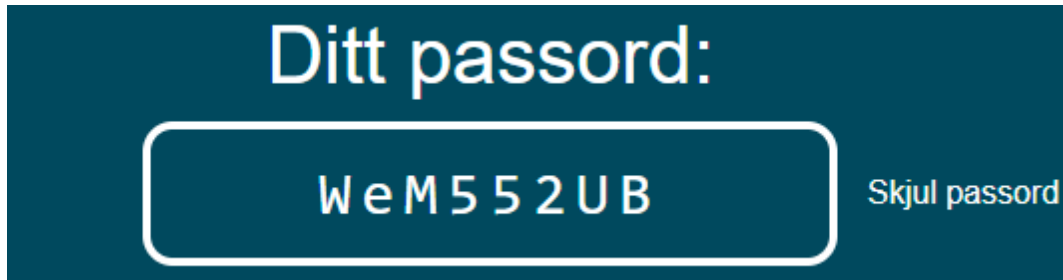
When clicking the button, we expect a “spinning wheel” to appear:



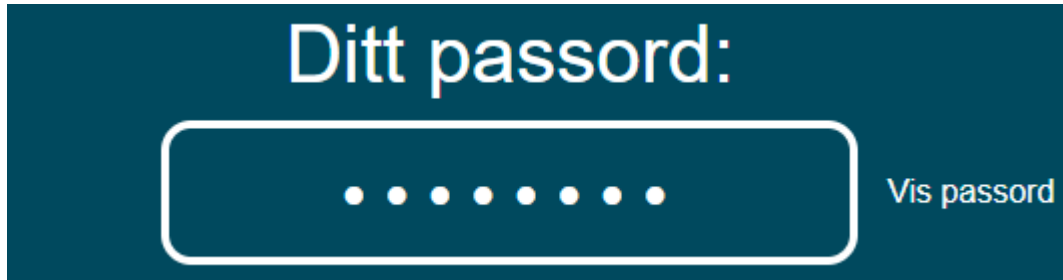
The graphic shown in the above picture depicts a section of the perimeter of a circle that is rotating at a constant pace. After this is done spinning (meaning the request has been processed and a response has been returned), we expect the following:



After clicking the “Show password” button on the right, we expect the new PSK to be displayed:



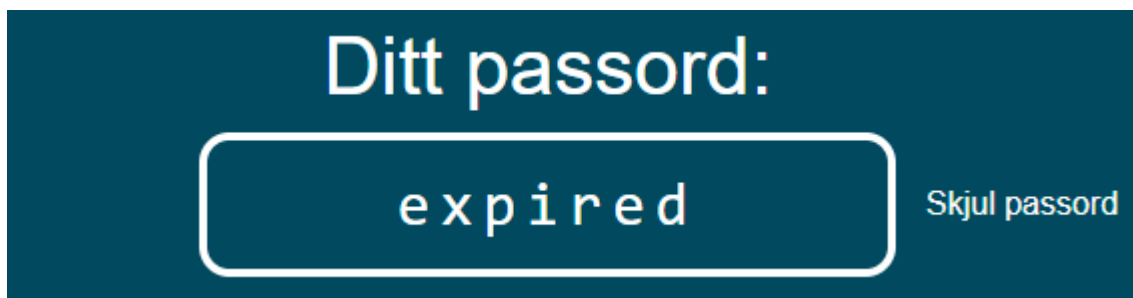
When clicking the button again, we expect the password to be hidden:



We expect an email to have arrived in the inbox of the email address associated with our Feide account:



Finally, we expect the password to read "expired" after five minutes:



We also perform variations of this test where, for example, we do not clear cookies and cache in the first step, do not choose delivery by email, or use different browsers.

No amount of testing will ensure the perpetual functioning of every aspect of the solution. It is therefore highly recommended that automatic error reporting is implemented using third-party solutions such as Sentry (<https://sentry.io>).

GDPR compliance

Our solution uses session keys to identify users. Because session keys are used to find the user's data (including their name, email address and Feide username), this is covered by the General Data Protection Regulation (GDPR):

(30): "Natural persons may be associated with online identifiers [...] such as internet protocol addresses, cookie identifiers or other identifiers [...]. This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them."

The requirement is that the user gives consent. When authenticating using Dataporten, the user will see a list of exactly what data is given to us, and the user may decline. Therefore, we consider the website GDPR compliant.

Discussion

This technology certainly brings interesting possibilities to the table. If we want to onboard IoT devices, the options are limited, and this solution to the problem is by far the best we have found.

We ask that the reader's imagination not be limited by the limitations of our proof of concept. Given a slightly larger team, more time and resources, a full-fledged solution suited for large-scale deployment is fully achievable.

On the other hand, emerging technologies may render this solution obsolete. This is by no means a certainty in the near future, but an "open" WPA3 network with a captive web portal for authentication may possibly be feasible. The remaining problem is that IoT devices cannot navigate captive web portals. WPA3, however, promises a simpler solution for devices with no display. We will see if this is feasible when the technology is ready, and whether or not the process becomes simpler or more complicated.

In this chapter, we will discuss the feasibility and future possibilities of the PPSK solution of which we have demonstrated a simple prototype.

The contents of this discussion will be summarized in the final report. We concluded that it is expedient to place the main discussion in this report, as we wish to limit the length of the final report.

Future possibilities

While we are satisfied with our product given the limited time and manpower of this project, there is much more we would want to do with this concept. Most such developments could be done simply and with significant returns.

First and foremost:

Admin dashboard

Let us examine the pieces of information we have available. We have:

- The Feide identity of the user
- Their PSK
- A name for the user's device(s)

HiveManager has an API endpoint for deleting users/PSKs.

This means that we could create these database tables for *users*, *devices*, and *registrations of devices by users*. The model for the latter, which represents the one-to-many or many-to-many association between devices and users, would hold the generated PSK.

This is all that is needed for a basic dashboard that shows which users have onboarded which devices. Furthermore, this dashboard could easily allow the administrator to perform actions against devices or users, such as deleting all devices by a user, or deleting a single device.

The state of the database could be checked by the authenticator. For example, we may use these database entries to determine if a user should be allowed to register another device. There may, for instance, be a restriction of 10 devices per user, which is not currently possible in our solution (because HiveManager sees one PSK as one user).

The creation of these database entries can easily be performed if the `finalize_response` method is overridden in a view inheriting from `lotConnectView`, or from within the `perform_generation` method of the custom ad hoc adapter.

A good place to start is admin site of any Django project. By default, this site is found at `host/admin/`. The Django admin site can be customized greatly. We can add custom actions here that can be selected from a drop-down list and executed with a button, for example.

User dashboard

Such a dashboard could in fact be made available to the user. Only the devices belonging to the user would be visible to the user, naturally. This would allow the user to remove devices in order to make room for new ones, change the description of devices, or reset the device's password, deleting the old one.

Custom emails

Another simply implemented improvement is to always tell HiveManager to not send the PSK by email to the user. Instead, our backend could send an email from our own mail server. This can also be implemented in the ad hoc adapter's `perform_generation` method.

As of now, the email containing the PSK is sent by Aerohive's servers and is only slightly customizable through HiveManager.

Device type restrictions

At a Uninett workshop we attended near the beginning of this project, representatives from Aruba talked about their product for Network Access Control and more called ClearPass. According to the representatives, their technology is capable of identifying the type of devices based on the contents of exchanged DHCP packets.

This technology would allow us to automatically approve or reject connections on the IoT network based on whether or not we believe the device to be compatible with eduroam (meaning it supports WPA2-Enterprise).

Another possible solution to this problem is to maintain a list of approved devices. In this case, the user would be asked to select the appropriate device. The MAC address of the connecting device could be compared to known MAC addresses of devices of this type, and

the truthfulness of the device type specification could be verified. This option may incur additional expenses and administrative work and requires a predictable MAC address pattern.

Serializers as validators

A more immediate improvement can be done to the Django project: Instead of programmatically checking for the presence of required values in the request (be it the authentication data or the generation options), it is possible to use serializers for this purpose. Serializers have the added benefit of easy type checking. Instead of implementing the function `validate_data`, we could make `serializer` a class attribute of `ValidatorMixin`. The serializer could look like this:

```
class GenerationOptionsValidator(serializers.Serializer):
    device_description = serializers.CharField(required=True,
                                              max_length=50,
                                              min_length=1)
    send_via_email = serializers.BooleanField(default=False)
```

The validity of the data may be checked, and a `ValidationError` may be automatically thrown if the data is not valid, using `some_serializer.is_valid(raise_exception=True)`.

WPA3

The coming WPA3 standard, which replaces WPA2, has a significant improvement in the security of personal and IoT networks. Its *Simultaneous Authentication of Equals* authentication protocol prevents two users from monitoring each other's traffic, even if they share the same Wi-Fi password. Combined with some other security mechanism that prevents users from accessing the network if they cannot provide Feide credentials, this technology may be able to achieve the same effect as PPSK — without the big investment. Furthermore, WPA3 promises to “simplify the process of configuring security for devices that have limited or no display interface” (Wi-Fi Alliance, 2018). This is worth researching as a separate project, as it may quickly render current multi-PSK/PPSK technology obsolete.

Feasibility

When dealing with enterprise networks, there is such a thing as something being too convenient sometimes. There is already a global network designed for research and education: eduroam. It was designed and is being managed by experts for security and reliability. It is greatly preferred that eduroam is used when it *can* be used. Sadly, because eduroam is based on authentication using digital certificates, not all devices can currently connect to it, and there is currently a need for this technology.

The convenience of our solution, however, means that a student is perhaps more likely to simply generate a few Wi-Fi passwords for their phones, tables and laptops. This is not the intention of our proposed solution; no one is better for it.

Without the immediate access to proprietary technology capable of categorizing connected devices based on their MAC address (as mentioned in the previous chapter), there may still be a solution that does not make the solution downright inconvenient.

In a typical educational institution, for instance, students should not be able to freely generate PSKs. Instead, educators at the faculty should be able to generate the necessary number of personal onboarding links, which are then handed out to students. Onboarding links take the form of a random, secret string, such as:

https://example.com/connect/NAiZ5iY0cY. These links can be handled by a common endpoint in Django using regular expressions. For example will the following...

```
path("?P<invitation_code>^[a-zA-Z0-9]*/$", ConnectView, name='connect')
```

... handle all possible combinations of alphanumeric strings. The actual string used would be checked against a database of pre-generated strings, and upon the first usage, it would be deleted from the database so that it is no longer valid. This can be done as simply as so:

```
from django.shortcuts import get_object_or_404

code = get_object_or_404(klass=InvitationCode, secret_code=invitation_code)
code.delete()
```

Additional invitations can be requested by individuals using a form, or by asking their faculty.

Any variation of this concept will likely be more feasible than the current solution, at least until solutions utilizing WPA3's Simultaneous Authentication of Equals are available.

It should also be mentioned that the system should be placed on a server that is only reachable from within the institution's private network (or using a VPN) before it is deployed. This way, abuse of the service (before the device is connected using the returned PSK) can be traced.

Before the user is allowed to request a PSK, they should be required to agree to a list of terms and policies. This agreement should be verifiable, meaning the date of the agreement, the user agreeing to the terms, and the version of the terms agreed to, should be stored in a database. This way, the terms need not be agreed to every time the user uses the service, but only when a new version of the terms is deployed.

In our opinion, the content of these terms should include the following at a minimum:

- The service may only be used to connect the user's own devices (or devices owned partially by the user in cases where the user is part of a team), and not devices belonging to others.
- Devices that support authentication using digital certificates, such as phones, tablets, and computers, should be connected using the standard means.
- The password issued should not be shared with anyone. The user should acknowledge that the password is considered an identifier, and any abuse carried out by devices connected using the password will be assumed to have originated from the user.

The ideal situation is that manufacturers of IoT devices realize that there is a market in enterprise networks and thus implement support for such networks. Even devices lacking displays could theoretically implement a mechanism for importing certificates, though it would be recommended that separate certificates be created for the devices belonging to a user. This way, an unattended IoT device with poor design in terms of security does not compromise the digital identity of the owner.

Amendment: Admin dashboard

This amendment was written on May 11, 2019.

While we discussed this feature in the *Discussion* chapter, we thought it was a shame to finalize the project without giving a concrete example of the proposed admin dashboard. Such an admin dashboard, in our mind, has the following two primary purposes: 1) Provide an overview of Feide users and their onboarded devices, and 2) allow administrators to delete or disable devices without directly interacting with HiveManager.

In order to achieve this, we created two models¹⁷: 1) FeideIdentity, and 2) DeviceRegistration. The former represents Feide accounts, as the name would suggest. The latter represents PSK requests that have been successfully processed.

FeideIdentity looks like this:

```
class FeideIdentity(models.Model):
    feide_user_id = models.CharField(
        unique=True,
        verbose_name="Feide ID",
        db_index=True,
        max_length=50,
    )

    name = models.CharField(
        max_length=50,
        verbose_name="name",
        db_index=True,
        blank=True,
    )

    email = models.EmailField(
        unique=True,
        blank=True,
    )
```

¹⁷ In Django, models are classes whose instances represent actual database entries. The model's attributes dictate the table's columns, while specific instances correspond to rows. Models are a very powerful and versatile feature of Django.

```

class Meta:
    verbose_name_plural = "Feide identities"

    def __str__(self):
        return self.feide_user_id + f" ({self.name})"

```

Its table contains three rows: 1) feide_user_id, 2) name, and 3) email. The values for these fields are retrieved from Dataporten. The meta class definition and the overriding of the `__str__` function affect how table rows will be displayed in the admin dashboard.

This is the DeviceRegistration model:

```

class DeviceRegistration(models.Model):
    tracker = FieldTracker(fields=['enabled'])

    feide_id = models.ForeignKey(
        FeideIdentity,
        on_delete=models.CASCADE,
        related_name='device_registrations',
        verbose_name="Feide identity",
    )

    hive_manager_id = models.CharField(
        max_length=80,
        verbose_name="HiveManager identity",
    )

    device_description = models.CharField(
        max_length=100,
        verbose_name="device description",
    )

    psk = models.CharField(
        max_length=20,
        verbose_name="PSK",
    )

    created_at = models.DateTimeField(
        auto_now_add=True,
        verbose_name="created at",
        db_index=True,
    )

    enabled = models.BooleanField(
        default=True,
        verbose_name="enabled"
    )

    def __str__(self):
        return f"{self.feide_id.__str__(): {self.device_description}"

```



```

def save(self, force_insert=False, force_update=False, using=None,
        update_fields=None):
    delete = self.tracker.has_changed('enabled') \
        and not self.enabled
    ret = super().save(force_insert, force_update, using,
                      update_fields)
    if delete:
        self.delete_from_hive_manager()

    return ret

def delete_from_hive_manager(self):
    url = "https://cloud-ie.aerohive.com/xapi/
          v1/identity/credentials"
    get_params = {'ownerId': OWNER_ID,
                  'ids': [],
                  'userName': self.hive_manager_id}
    hive_manager_user = attempt_json_loads(
        requests.get(url=url, params=get_params, headers=HEADERS)
        )._content['data'][0]
    real_id = hive_manager_user['id']

    post_params = {'ownerId': OWNER_ID, 'ids': [real_id]}
    response = requests.delete(url=url, params=post_params,
                              headers=HEADERS)

    if not response.status_code == 200:
        raise ValueError("HiveManager did not return a 200 OK
                          response")

```

It declares a *tracker*, which keeps track of changes to field values. Specifically, we are interested in whether the value of `enabled` changes over time. The reasoning will become apparent.

In addition to a series of fields, it declares a foreign key to the `FeidIdentity` model. The argument passed to the `on_delete` parameter dictates what should happen to the `DeviceRegistration` when the `FeidIdentity` it is related to is deleted. *CASCADE* dictates that the `DeviceRegistration` should also be deleted. The value of `related_name` dictates how a reverse lookup is performed. By setting it to “`device_registrations`”, we are able to find all the related `DeviceRegistration` objects using the `device_registrations` attribute that is created for the `FeidIdentity` model.

The `save` method of a model is called when an instance is created or updated (unless it is updated or bulk-created using a `QuerySet`, but that is beyond the scope of this document). This `save` method checks if the registration has been disabled by consulting with the field `tracker`. If so, the `delete_from_hive_manager` method is called.

`delete_from_hive_manager` works similarly to the `HiveManagerAdapter` class, but performs two requests (neither of which is a POST request). The first request queries `HiveManager` in order to get a “list” of identities matching the username. Because we add a

timestamp to the username, we can be confident that it is unique; only one result will be found. We retrieve the `id` value of this result. We need this value in order to delete the user.

Keep in mind that a *user* in this context really means a *registered device* in HiveManager.

Next, a DELETE request is sent to the same endpoint. The *id* of the user is provided in the query parameters. This results in the user being deleted from HiveManager, rendering the user's PSK invalid.

These models were defined in the *models* module of the *connect* app.

The creation of these models results in a database state change. Such changes are kept track of using what is called *migrations* in Django. These are files with instructions on exactly how the database should be altered to be run in sequence. These migrations can be found in the *migrations* package of the *connect* app.

Django features built-in templates and frameworks specifically for admin dashboards that simplify administrative tasks. Therefore, we only need to perform minimal customization of the admin dashboard. We decide which models should be featured on the dashboard by creating *model admin* classes and registering them. This is how we did it:

In a module named *admin.py* in the *connect* app, we first added the following class:

```
@admin.register(FeideIdentity)
class FeideIdentityAdmin(admin.ModelAdmin):
    list_display = ['feide_user_id', 'name']
    search_fields = ['feide_user_id', 'name']

    @staticmethod
    def disable_all_devices(_instance, _request, queryset):
        for feide_identity in queryset:
            for device_registration in \
                feide_identity.device_registrations.all():
                if device_registration.enabled:
                    device_registration.enabled = False
                    device_registration.save()

    actions = ('disable_all_devices',)
```

The `@admin.register` decorator tells Django that we wish to add this model admin to the dashboard. The argument, `FeideIdentity`, tells Django that this model admin will be used to manage the `FeideIdentity` table.

The dashboard allows us to see objects in a list, or individually in a more detailed manner. The `list_display` attribute says which attributes should be displayed in the list. The `search_fields` attribute says which fields should be searchable.

We defined a `disable_all_devices` method, which (as all Django admin actions) accepts, among other things, a `queryset`. This `queryset` is a representation of an SQL query

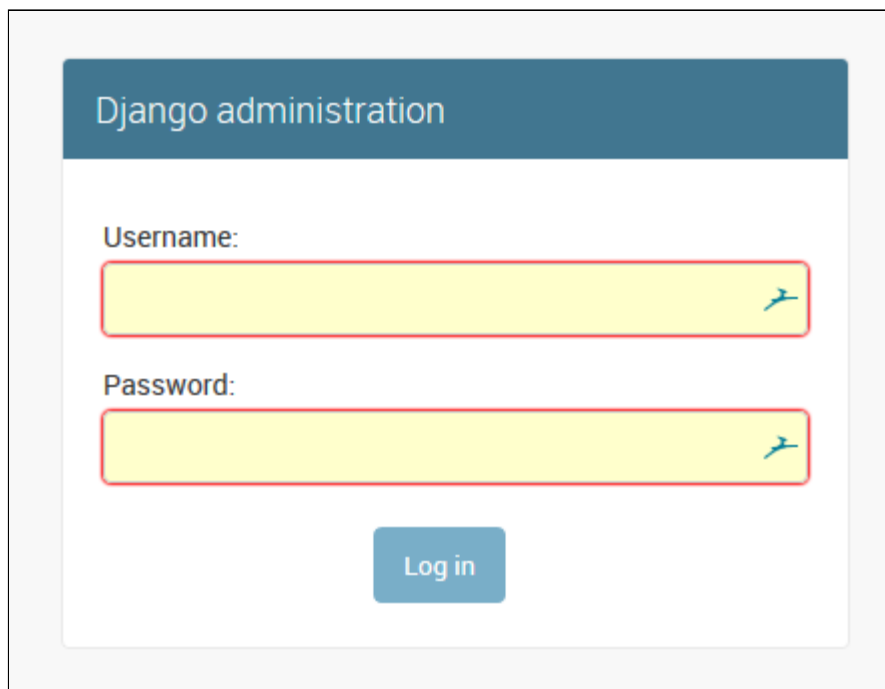
which will be lazily executed (meaning it will be executed when the results are accessed for the first time). This queryset contains all the rows that were selected before this action was executed. For each FeideIdentity that was selected, we iterate through the related DeviceRegistrations. If the DeviceRegistration is not already disabled, we disable it and save it. When saving it, we trigger the `delete_from_hive_manager` method as described.

We add this action to the dashboard by setting the `actions` attribute, including the method.

The admin dashboard can be accessed, by default, at `hostname.tld/admin/`. In our case, it is located at `api.environmentname.tld/admin/`. We sign in using a superuser, which we created using the following command:

```
python manage.py createsuperuser
--settings=uninett_api.settings._environment
```

This is the login screen:



Here is a section of the page we are presented with after authorization:



Please ignore the presence of *Device registrations* for now.

Clicking on *Feide identities*, we can see the following:

Select feide identity to change

ADD FEIDE IDENTITY +

Q Search

Action: Go

0 of 1 selected

<input type="checkbox"/>	FEIDE ID	NAME
<input type="checkbox"/>	feide:redacted	Magnus Magpie Bakke

1 feide identity

The Feide identity was redacted before the screenshot was made. If we click on the entry, we can also see the user's email address as well.

If we select rows and choose an action of the *Action* list, we can perform the following actions:

Select feide identity to change

ADD FEIDE IDENTITY +

Q Search

Action: Go

1 of 1 selected

<input type="checkbox"/>	FEIDE ID	NAME
<input checked="" type="checkbox"/>	feide:redacted	Magnus Magpie Bakke

1 feide identity

Dropdown menu options:

- Delete selected Feide identities
- Disable all devices

Both will have the same effect (due to a *signal receiver* that handles the deletion of objects, which will be explained shortly), but the *Disable all devices* option does not delete any objects: It only marks them as disabled.

Before we continue, we should explain the following:

- How the identity was created,
- How we added the *Device registration* model admin, and
- How the deletion of these objects is handled.

As mentioned in the *Future possibilities* subchapter of *Discussion*, the creation of any necessary database objects can be performed in the `finalize_response` method of `ConnectView`. We have overridden this method thusly:

```
def finalize_response(self, request, response, *args, **kwargs):
    if response.status_code == status.HTTP_201_CREATED:
        hive_manager_id = response.data['username']
        response.data = response.data['psk']
        session = Session.objects.get(
            pk=self.authentication_data['session_key']
        ).get_decoded()
        user_data = get_user_data(
            access_token=session['access_token']
        )
        name = user_data['name']
        email = user_data['email']
        feide_username = user_data['userid_sec'][0]

        feide_identity, _created = FeideIdentity.objects.get_or_create(
            feide_user_id=feide_username
        )
        if feide_identity.name != name \
            or feide_identity.email != email:
            feide_identity.name = name
            feide_identity.email = email
            feide_identity.save()

        device_description = self.generation_options['device_type']
        psk = response.data
        DeviceRegistration.objects.create(
            feide_id=feide_identity,
            device_description=device_description,
            psk=psk,
            hive_manager_id=hive_manager_id
        )

    return super().finalize_response(request, response,
                                     *args, **kwargs)
```

It should be mentioned at this point that we had the `HiveManagerAdapter` return a response in the following format:

```
{
    "psk": "xxxxxxxx",
    "username": "username: yymddhhMMss"
```

```
}
```

This is in place of the previous response format (containing only the PSK):

```
"xxxxxxxx"
```

The method only performs any logic if the response's status code is 201 CREATED. We do not wish to create DeviceRegistration objects if the device was not successfully registered. First, it extracts the *username* value of the response data before replacing the response data with only the PSK. In this way, we have retrieved the HiveManager username without altering the final response.

Next, it gets the user data from the user's session. We are interested in the user's name, email address and Feide identity. These will be stored in a FeideIdentity object.

The following statement creates a new FeideIdentity with the provided `feide_user_id` value unless it already exists (we do not want to create duplicate identities):

```
feide_identity, _created = FeideIdentity.objects.get_or_create(  
    feide_user_id=feide_username  
)
```

We do not care whether or not the identity already exists.

If this identity's stored user data differs from the data just retrieved from Dataporten, we update the identity with the recently retrieved data.

Next, we get the device description from the `generation_options` (which was gotten from the request in the `dispatch` method). We get the PSK from the response object. These values will be used to populate the fields of a new DeviceRegistration object. We now have all the required information for this object:

- The FeideIdentity object (for the foreign key),
- the device description (from the generation options),
- the PSK (from the response data),
- the HiveManager username (extracted from the adapter response), and
- the timestamp of creation (now).

The timestamp will be automatically set.

We create the DeviceRegistration object thusly:

```
DeviceRegistration.objects.create(  
    feide_id=feide_identity,  
    device_description=device_description,  
    psk=psk,  
    hive_manager_id=hive_manager_id  
)
```

Finally, the modified response (containing only the PSK as expected by the frontend) is returned.

In summary, this function is called every time a user successfully generates a PSK. It creates an identity for them in the database (unless it already exists) and creates a `DeviceRegistration` object that represents their device and the details about its creation.

The `DeviceRegistration` model is managed using the following model admin:

```
@admin.register(DeviceRegistration)
class DeviceRegistrationAdmin(admin.ModelAdmin):
    list_display = ['created_at', 'feide_id', 'get_name',
                   'get_feide_user_id', 'device_description', 'enabled']
    list_select_related = True
    list_filter = ['enabled']
    search_fields = ['feide_id', 'feide_id__name',
                    'feide_id__feide_user_id']

    @staticmethod
    def disable(_instance, _request, queryset):
        for device_registration in queryset:
            if device_registration.enabled:
                device_registration.enabled = False
                device_registration.save()

    actions = ('disable',)

    @staticmethod
    def get_name(obj):
        return obj.feide_id.name

    @staticmethod
    def get_feide_user_id(obj):
        return obj.feide_id.feide_user_id
```

We will only summarize what it does.

The `list_display` attribute accepts not only database column names, but also function names. Because we wish to display values of a related object (the `FeideIdentity`), we must add the `get_name` and `get_feide_user_id` functions, which first retrieve the related `FeideIdentity` object before returning that object's `name` or `feide_user_id` values.

We also wish to allow the administrator to filter the results based on whether or not the device is disabled:

```
list_filter = ['enabled']
```

The `disable` method, which is added as an action, iterates through the selected `DeviceRegistration` objects and disables them before saving them. This deletes their `HiveManager` entries, rendering the PSKs unusable.

Here, we can see a list of devices that we are about to delete:

Action: <input type="text" value="-----"/> <input type="button" value="Go"/> 3 of 6 selected		GET NAME	GET FEIDE USER ID	DEVICE DESCRIPTION	ENABLED
<input type="checkbox"/>	CRE				
Delete selected device registrations					
<input checked="" type="checkbox"/>	May 10, 2019, 10:12 p.m.	feide:magnbakk@ntnu.no (Magnus Magpie Bakke)	Magnus Magpie Bakke	Test	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	May 10, 2019, 10:12 p.m.	feide:magnbakk@ntnu.no (Magnus Magpie Bakke)	Magnus Magpie Bakke	Støvsuger	<input checked="" type="checkbox"/>
<input type="checkbox"/>	May 10, 2019, 9:38 p.m.	feide:magnbakk@ntnu.no (Magnus Magpie Bakke)	Magnus Magpie Bakke	test	<input checked="" type="checkbox"/>
<input type="checkbox"/>	May 10, 2019, 8:16 p.m.	feide:magnbakk@ntnu.no (Magnus Magpie Bakke)	Magnus Magpie Bakke	Støvsuger	<input type="checkbox"/>
<input type="checkbox"/>	May 10, 2019, 6:30 p.m.	feide:magnbakk@ntnu.no (Magnus Magpie Bakke)	Magnus Magpie Bakke	Støvsuger	<input type="checkbox"/>
6 device registrations					

After the action has been executed, the devices are no longer enabled:

DEVICE DESCRIPTION	ENABLED
Test	<input type="checkbox"/>
Test	<input type="checkbox"/>
Støvsuger	<input type="checkbox"/>

Finally, we must handle the deletion of objects. It should be a universal truth that, if a DeviceRegistration does not exist, the corresponding PSK should no longer be valid. Therefore, we need to catch the deletion event of these objects and call the `delete_from_hive_manager` method before the objects are deleted from the database. This can be achieved using signals.

Django implements many signals, but we are only interested in handling the `pre_delete` signal. In a `signals` package, we created a `receivers.py` module. This module is imported in the `connect` app's AppConfig:

```
class ConnectConfig(AppConfig):
    name = 'connect'

    def ready(self):
        from connect.signals.receivers import . # noqa
```

The `# noqa` comment simply instructs the editor and any code inspection software to ignore the occurrence of an unused `import` statement, as it serves the purpose of making the receivers accessible.

In this module, the following receiver is defined:

```
@receiver(pre_delete, sender=DeviceRegistration)
def deleting_device_registration(instance: DeviceRegistration, **kwargs):
    instance.delete_from_hive_manager()
```


This method is called before a DeviceRegistration object is deleted from the database. Note that there is no need to make a receiver for Feididentity on account of the use of the *CASCADE* scheme, which ensures that all DeviceRegistrations pointing to the deleted Feididentity is also deleted.

Deviations from the design report

Unnecessary security feature

We considered adding an optional random waiting period before the PSK is generated. This would make sense if the the timestamp of the request could be used to guess the PSK with reasonable accuracy, as would be the case if the random number generator used to generate PSKs was seeded with the current time (often a default setting for random number generators). However, because of the naturally unpredictable delay when sending requests from the frontend to the backend, and from the backend to HiveManager, it should not be possible to do this. We also trust that HiveManager uses a highly secure random number generator.

Email input field

The frontend designs featured an input field for the user's email address. Uninett has asked that we do not allow the user to specify an email address. Instead, we use the email address associated with the Feide user.

Exception cases

The design report proposed that we should handle the following errors:

- 400 BAD REQUEST
- 401 UNAUTHORIZED
- 403 FORBIDDEN
- 500 INTERNAL SERVER ERROR

In addition to these, we must handle the following:

Case or status code	Meaning	Resolution
408	The request timed out. May occur if the server has crashed or is down.	Display an appropriate message.
404	Not found. May occur if the Django application is down.	Display an appropriate message.
503	Service unavailable. Returned if	Display an appropriate

	HiveManager or Dataporten is not responding.	message.
Unknown browser problem	The browser probably does not support CORS.	Ask the user to try a different browser.

Reachability of the server

In the design report, it was mentioned that the frontend will be reachable from within the network. It is, in fact, reachable on the open internet, because we migrated to Amazon Web Server instances. This was more convenient for testing purposes, and gave us greater control over the servers.

Checking of support for cookies

The design report postulated that "... our Feide-specific solution will require additional views that handle, among other things, the verification that the client supports and accepts the storing of cookies." Because we pass the session key using query strings instead of using the conventional means of storing and retrieving sessions, this is no longer a requirement.

Conclusion

After our research into possible solutions to the issue of connecting IoT devices to wireless networks securely, we have developed designs for a prototype which we have now developed into a working proof of concept. The results show that the solution is indeed viable, assuming the necessary alterations and additional functionality are included in the final implementation. The technology has been largely unproblematic.

During the execution stage, both candidates found themselves occupied with education and work, meaning every other Monday, most Wednesdays and nearly every weekend were spent working on the project. Work was done before, during and after staying in Uninett's offices. This is a considerable workload that should not be taken lightly, especially in conjunction with employment. We hope that it has produced a satisfactory result that is helpful to Uninett and whomever else it may concern. Nevertheless, we have found a new appreciation for the importance of coordination, thoroughness and embracing new and emerging technologies.

The conclusion of this report marks the end of the execution stage. The three reports produced thus far will be accompanied by a final report that reflects on the entirety of the project and summarizes various aspects of the planned solution and the actual solution developed.

Bibliography

Tuomi, J. (2018, October 9). Getting a React + Django App to Production on DigitalOcean with Travis, Caddy and Gunicorn. Retrieved April 19, 2019, from https://medium.com/@jans.tuomi/getting-a-react-django-app-to-production-on-digitalocean-with-travis-caddy-and-gunicorn-5c397383fcd5?fbclid=IwAR0PXHp3lph0VWYH3iJ6ep6HN_u0qWQcYwoZzGxxOxQXBhsQ5uQbJr6Crok

Wi-Fi Alliance. (2018, January 8). Wi-Fi Alliance® introduces security enhancements | Wi-Fi Alliance. Retrieved April 25, 2019, from <https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-security-enhancements>

Aruba Networks. (n.d.). Secure Access with Aruba ClearPass: BYOD Network Security Solutions [Product page]. Retrieved May 5, 2019, from <https://www.arubanetworks.com/products/security/network-access-control/secure-access/>

European Parliament. (2016, April 27). REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL (General Data Protection Regulation). Retrieved May 7, 2019, from <https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679>

AUTHENTICATION IN THE INTERNET OF THINGS

Final report

Part of a Bachelor's thesis

**Presented to the Institute of computer technology and informatics
of the Norwegian University of Science and Technology
by Magnus Bakke & Liang Zhu**

Submitted in partial fulfillment for Bachelor's degree of
Informatics with specialization in network administration
during the year 2016–2019

Preface

We, the authors — Magnus Bakke and Liang Zhu — are students of the Norwegian University of Science and Technology (NTNU), where we study Informatics with specialization in systems administration. We are pleased to present this final report, which marks the end of the finalization stage and indeed this Bachelor's thesis. After 19 weeks of working on our thesis, we have learned more in such a short amount of time than ever before. We could not have hoped for a more relevant and exciting assignment.

This document succeeds three previous reports: The research report, the design report, and the execution report. We recommend reading these documents in the stated order if you have not already and the contents of this document interests you.

We hope our hard work will be useful.

Acknowledgements

We would like to thank to acknowledge all those who have helped and supported us.

First and foremost, we would like to extend our gratitude to Uninett AS — especially Jørn Åne de Jong, Tom Ivar Myren and Otto Wittner — for approving our request to write our thesis on their assignment *Authentication in the Internet of Things*, and for giving us guidance, direction and office space throughout the project period.

Stein Meisingseth of NTNU has been an excellent supervisor who has provided structure, encouragement and advice that we have benefited greatly from.

We would like to sincerely thank Aerohive Networks for taking an interest in our project and lending us integral hardware. Special thanks go to Jonas Mellander of Aerohive Networks for a highly informative demonstration of HiveManager, and to Fanny Carlson for responding to and approving our request, and for coordinating our communications with Aerohive.

We are grateful for the interest taken in our project by Anders Lagerqvist and Tore Henriksen of Aruba.

Finally, we would like to thank Vidar Kværnø Stokke for his interest in our project, and for providing us with hardware for testing.

Contents

Preface	2
Acknowledgements	2
Contents	3
Case summary	4
Method	5
Execution	7
Future developments	10

Case summary

Demand for IoT devices in enterprise settings is on the rise. Therefore, Uninett AS posted an assignment named *Authentication in the Internet of Things*. This assignment caught our attention as an opportunity to learn about the security problems of IoT and how they might be solved. Uninett approved our application.

Uninett AS is the state-owned company that provides Norwegian universities, university colleges and research institutions with internet access and various IT services. We were supervised by Systems Developer Jørn Åne de Jong of Uninett, and University Lecturer Stein Meisingseth of the Norwegian University of Science and Technology. Senior Advisor Tom Ivar Myren and Otto Wittner of Uninett were also heavily involved in the project.

Before going into further detail, we must illuminate the problem. We make the distinction between *personal* networks and *enterprise* wireless networks.

Personal networks are typically used in homes, as well as in many restaurants, cafés and similar establishments. When connecting to a personal network, the owner of the device to be connected asks the owner of the wireless network for the password. If the owner of the network trusts the guest, the password is shared and the guest may connect his or her devices. This Wi-Fi mode is called WPA2-Enterprise.

A consequence of how Wi-Fi works (and must work) is that this password, or pre-shared key (PSK), acts as the encryption key for communications, meaning that anyone in possession of the PSK is able to decrypt the communications of others using the same network. In larger enterprise settings, this model of trust collapses. In an organization with dozens or more employees, for example, we cannot assume that the PSK will not be shared with unauthorized persons, and we cannot know the intentions of each individual. Instead, such networks should use the WPA2-Enterprise mode. WPA2-Enterprise uses public-key cryptography for authentication and initial encryption and communication. Using this encryption scheme, a unique, dynamically generated encryption key can be communicated, and this key can be used for further communication. This ensures that no user is able to monitor another user's communications. This also acts as a strong authentication scheme.

The problem is rooted in the fact that IoT devices typically do not support WPA2-Enterprise. They often only allow the user to supply a password and preferred SSID. The consequence is that a wireless network containing IoT devices cannot use the secure WPA2-Enterprise mode of WPA2.

The assignment asked us to investigate options for authentication infrastructure in IoT applications. This involves conducting a survey of available solutions and technologies and, if possible, identifying and analyzing solutions that are suitable for deployment and use in the Norwegian academic network. The assignment calls for the development of a proof of concept. Finally, the feasibility of the proposed solution should be analyzed.

Method

We prepared several standards, guidelines and goals to aid us in producing a result of satisfactory quality within the given limitations.

It was decided that we should hold a meeting roughly every weeks. These biweekly meetings each produced meeting minutes, which will be presented to the stakeholders. The meetings were an integral part of our method as the attendees often provided valuable feedback, advice and opinions which would influence subsequent decisions.

Furthermore, it was decided that a summary of activities should be written for each day of work. These summaries were included in weekly reports, which have been presented to the stakeholders.

A progress plan for the week was developed at the beginning of each week. These plans dictated which activities would be conducted during the week.

For the purpose of giving ourselves a better overview of the work that has been done and still remains to be done, we decided to require that we use a modified version of Scrum. As the work on tasks progressed, sticky notes representing the tasks were moved to their appropriate category (*open, in progress, review, or done*).

We planned four stages for the project: 1) The research stage, 2) the design stage, 3) the execution stage, and 4) the finalization stage. These stages produced the following documents respectively:

1. The prestudy report
2. The design report
3. The execution report
4. The final report

We defined goals and a purpose for each stage in the prestudy report. Briefly summarized, the purpose of each stage was declared thusly:

Stage	Purpose
The research stage	To conduct a survey of existing technologies, analyze existing technologies, propose a best suited solution, minimize risk, analyze financial feasibility, prepare the candidates for subsequent stages, guide decisions on whether or not to continue down the current path, and produce the prestudy report.
The design stage	To develop detailed plans and designs for how a proof of concept should be developed, look and feel; to identify and correct problematic designs, highlight potential security issues that need to be addressed, and produce the design report.
The execution stage	To develop the proof of concept according to designs; to test,

optimize, and extend the developed proof of concept according to goals and requirements, and to produce the execution report.

The finalization stage To make any necessary amendments to previous reports, ensure correct formatting, prepare documents for submission, evaluate our achievements compared to goals and requirements defined in the research stage, and to summarize these findings, as well as the entirety of the project (briefly), in the final report.

The information collected in the research stage exists solely on the internet. Sources for information and statements that are not common knowledge and accepted facts have been cited. Sources were found with the help of search engines, and relevant sources were compiled for later usage.

The prestudy report included a risk analysis, which affected the scope of the subsequent stages. Specifically, it reduced the scope in order to minimize the risk of overly ambitious designs.

The hours spent working on the project was recorded in a spreadsheet for each candidate. Hours were grouped by date, and each date was accompanied by a short description of the nature of the activities. These spreadsheets will be presented to the University.

An agreement between Uninett AS, the candidates, and NTNU was prepared and signed by these acting parties. This agreement specified how the product and reports may be used by Uninett, the students, NTNU and external stakeholders.

The candidates divided work amongst themselves based on whom is more qualified to complete the task. In general, the candidates would agree on which tasks must be completed (according to the progress plan and Scrum board), and each candidate would complete one task each simultaneously. Guidance was provided by the other candidate when needed.

Execution

Throughout the three first stages of the project, we researched, designed and developed what we perceive to be the optimal possible solution.

We set up a testing server and a production server running Ubuntu 18.04 LTS for the proof of concept. On these servers, we installed Caddy and gunicorn. Caddy serves HTTP requests, and gunicorn proxies these requests to a Django REST framework project, which we will refer to as the backend. Caddy also serves a static HTML document with JavaScript that we will refer to as the frontend.

The solution is summarized thusly:

When users visit the frontend, they are redirected to Dataporten, our Identity Provider, for authentication on their Single Sign-On (SSO) page, unless they are already authenticated. After authenticating on the SSO, the user is redirected to the backend, where data about the Dataporten/Feide account is stored in the user's session. This data includes the user's name, email address, and a unique Feide identifier. The user is then redirected back to the frontend. This time, the user is found to be authenticated, the form containing input fields for the device description and a checkbox for the option of email delivery is shown. When the user has filled out the form, and validation has passed, the user may click a button to get a new PSK for the IoT wireless network.

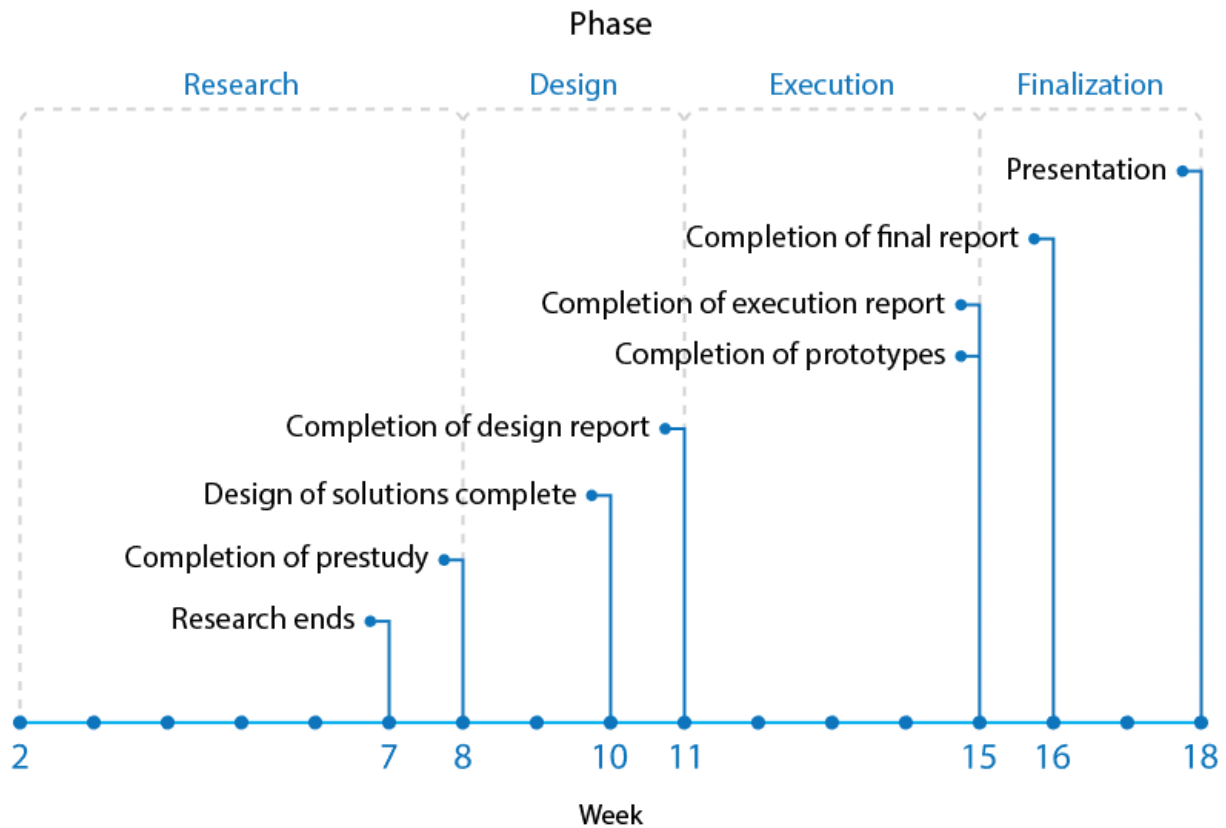
The backend, which was built using the Python framework Django REST framework, accepts HTTP requests from the frontend, or rather from the user's web browser. The backend expects the request to contain authentication data (a session key) and generation options (including a description of the device to be connected, as well as a Boolean value indicating whether or not the user wants the PSK to be delivered by email).

The authentication data (session key) is used to retrieve the user data from the user's session. Dataporten is queried to verify that the authentication is not expired.

After successful authentication, the generation options data in the request body is formatted and sent to HiveManager's API, which generates a new PSK and returns it to the backend. This PSK, along with the Feide user data, is stored in a FeideIdentity object in the database. A DeviceRegistration object is created as well, and the PSK and generation options is stored in it. The Feide identity that registered the device is also recorded here. This allows administrators to view registered identities and associated devices in what we call the administrative dashboard, which allows the administrator to disable some or all devices of any user.

The PSK is returned to the frontend and displayed to the user. If the user selected the email delivery option, the PSK is also delivered via email.

The development of this proof of concept required extensive research, planning and testing. In the research stage (which produced the prestudy report), we developed the following progress plan:



This progress plan has mostly been followed. However, the prestudy report also included a goal that stated that there should be no deviations from the progress plan exceeding three days.

This report is being written in week 20. In other words, there is a four week discrepancy between the progress plan and the actual process. A similar delay is recorded for the execution report. These delays were caused by unexpected difficulties.

Another goal stated that the project should be completed at least three days before the deadline of May 20th. At the time of writing, the date reads May 19th. We would like to stress that these goals are ideals, and not critical to the success of the project, unlike the functional and non-functional requirements.

Except for these two process goals, the remaining process goals were met. These are the remaining process goals:

- 500 (+/- 5%) hours spent per candidate
- Zero disputes/conflicts between the candidates
- Documentation of every major activity and decision

Both candidates are well within the 500 (+/- 5%) hour requirement. We have experienced zero disputes or conflicts; we have experienced harmony in direction, activities, decisions and goals. Every major decision has been documented in the research, design and execution reports, as well as meeting minutes and weekly reports.

Regarding the performance goals, these were met as well:

- Achieve internet connectivity for IoT devices in less than two minutes
- Achieve an average upload and download speeds greater than 80% of those normally achieved on Uninett's network
- Experience no more than 120% the amount of drops to less than 20% of the average download/upload speed compared to a non-IoT connection

It is worth mentioning that the latter two goals were not tested continuously. Speedtests were performed twice. Because we did not proceed with the "hotspot solution", which was proposed as an alternative in the prestudy report, we did not anticipate noticeable degradation in performance. We frequently generated PSKs and connected devices in less than 30 seconds.

The prestudy report also included a risk analysis, which concluded that we should maintain a high degree of minimalism in order to mitigate the risk of overly ambitious designs. As a result, we have been able to complete the proof of concept within the time frame of the project. We have sacrificed many proposed functions and use cases (as described in the *Future developments* chapter) in order to develop a minimum viable product. We believe that demonstrating the feasibility of the solution is more important than approaching a solution that is truly ready for full scale deployment. Perhaps as a result of our risk analysis, we have not encountered any emergencies.

The following (all) functional requirements have been met:

- Users may bring their own IoT devices to the premises and achieve internet connectivity
- The user responsible for abuse should be identifiable
- The communication between a device and the access point should be encrypted and visible only to the user who connected the device
- The solution must allow the user to authenticate using their institutional credentials
- Each user must be given a unique PSK
- PSKs must be generated automatically on demand
- Roaming between access points
- A custom dashboard for requesting PSKs

Additionally, the following (all) non-functional requirements have been met:

- The solution should be financially feasible
- The solution should not cause harmful interference
- The solution should not cause excessive administrative work
- The solution should not require significant changes to network architecture
- The solution needs not be ready for large-scale deployment, but it should be easy to prepare for such deployment

The prestudy report stated that the reports should preferably be formatted using LaTeX. We did not find time for this. This was not a critical requirement.

In summary, we developed almost exactly the planned solution. The only deviations from the initial plans pertain to process goals. Due to unexpected problems — especially the configuration of Apache (which we later abandoned in favor of Caddy) — resulted in significant delays. Therefore, we were unable to achieve the ideal goal of delays of no more than three days.

Future developments

In the execution report, we discussed several possible developments that could make the solution better suited for large scale deployment.

First and foremost, we propose extended functionality for the administrative dashboard. Currently, administrators can sign in to the admin dashboard and view registered users and devices. A minor change to the admin class (please see the execution report for more details) would provide a better overview of which devices are registered by which users. Furthermore, it would be beneficial to record and maintain the updated IP address of the device at all times, so that the owner of a device found to exhibit malicious behavior could be identified without using HiveManager.

By recording the identities and devices of a user, the authenticator could inspect the database when deciding whether or not the user should be authorized to generate a PSK. One condition that could be required is that the user has fewer than ten devices, for example.

The execution report also recommended that a dashboard be developed for users as well, so that they can view and manage their own devices. In conjunction with a limitation on the number of PSKs a user may generate, the user may remove existing devices to make room for new ones. This gives the user an incentive to remove unused PSKs.

The recording of identities and device associations also enables us to delete PSKs associated with past students or employees who are no longer affiliated with the organization.

It would be desirable to implement restrictions regarding which devices are allowed to be connected to the IoT network. Aruba implements technology that they claim is capable of identifying the type of connected devices. The possibility of using this technology to restrict which devices may be connected to the IoT network should be investigated. This is desirable because the proof of concept makes it convenient to choose the IoT network instead of eduoam.

The execution report also recommended using a different email server. Currently, the PSK (if delivered by email) is sent from HiveManager. It is beneficial to separate the PSK provider from the user entirely; the user should not be aware of which service is being used. This decoupling allows us to switch providers without users noticing, and also allows us to customize emails fully, and to send the email containing the PSK from an institutional email address instead of one owned by Aerohive.

Lastly, we recommend investigating possible solutions based on WPA3. If traceability is not a priority, WPA3 can provide end-to-end encryption that ensures the confidentiality of communications, even on “open” networks.

