



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

**WEB APPLICATION FOR CERTIFICATE MANAGE-
MENT**

WEBOVÁ APLIKACE PRO SPRÁVU CERTIFIKÁTŮ

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

SIMON KOBYDA

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. RADEK BURGET, Ph.D.

BRNO 2020

Bachelor's Thesis Specification



Student: **Kobyda Simon**
Programme: Information Technology
Title: **Web Application for Certificate Management**
Category: Web
Assignment:

1. Study current technologies for creating applications with a thick web client focusing on React.js and related technologies.
2. Learn about Cockpit project, its architecture, and the way of plugin development.
3. Design a Cockpit plugin for managing Linux certificates via the web interface.
4. Implement the proposed solution using appropriate technologies. Also implement a set of tests to verify the functionality of the plugin.
5. Perform testing of the implemented solution.
6. Evaluate the results.

Recommended literature:

- Banks, A.: Learning React: Functional Web Development with React and Redux, O'Reilly, 2017
- Projekt Cockpit: <https://cockpit-project.org/>

Requirements for the first semester:

- Items 1 to 3.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Burget Radek, Ing., Ph.D.**
Consultant: Papadourakis George, prof., TEI of Crete
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: April 20, 2020

Abstract

The aim of this work is to create software to request and manage X.509 certificates. Its main emphasis is to provide a smart and intuitive user interface, which would enable administrators, who do not have much knowledge about certificate, to request and manage them. The work is implemented using React as a plugin for Cockpit, which by extension will fit it into a bigger picture of Red Hat's portfolio.

Abstrakt

Cielom tejto práce je vytvorenie softvéru na získanie a správu X.509 certifikátov. Hlavný dôraz je na poskytnutie múdreho a intuitívneho užívateľského rozhrania, ktoré by dovolilo spravovať certifikáty aj administrátorom, ktorí nemajú o nich veľa znalostí. Práca je implementovaná s technológiou React ako plugin pre Cockpit, vďaka čomu si nájde svoje miesto aj vo väčšom obraze Red Hat portfólia.

Keywords

certificates, web, application, Cockpit, Red Hat, SSL, TLS, X.509, user interface, React

Klíčová slova

certifikáty, web, aplikácia, Cockpit, Red Hat, SSL, TLS, X.509, užívateľské rozhranie, React

Reference

KOBYDA, Simon. *Web Application for Certificate Management*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Burget, Ph.D.

Web Application for Certificate Management

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Radek Burget, Ph.D. The supplementary information was provided by Matej Marušák from Red Hat and consultant George M. Papadourakis from Hellenic Mediterranean University. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Simon Kobyda
July 30, 2020

Acknowledgements

I would like to give special thanks to my supervisor Ing. Radek Burget, Ph.D., consultant at Red Hat Matej Marušák and consultant George M. Papadourakis from Hellenic Mediterranean University.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 1.1 | Public key infrastructure | 3 |
| 1.1.1 | Certificates | 4 |
| 1.2 | Cockpit and problems of Linux administration | 6 |
| 2 | Technologies | 7 |
| 2.1 | Javascript | 7 |
| 2.2 | React | 7 |
| 2.2.1 | Javascript XML | 7 |
| 2.2.2 | Stateless components | 8 |
| 2.2.3 | Stateful components | 8 |
| 2.3 | Patternfly | 9 |
| 2.3.1 | Fundamentals | 10 |
| 2.3.2 | Designer Guidelines | 10 |
| 2.4 | D-Bus | 11 |
| 2.4.1 | Objects | 11 |
| 2.4.2 | Messages | 12 |
| 2.5 | Certmonger | 12 |
| 3 | Design | 14 |
| 3.1 | Listing certificates | 14 |
| 3.2 | Requesting a certificate | 16 |
| 3.3 | Importing a certificate | 17 |
| 3.4 | Resubmitting a certificate | 17 |
| 3.5 | Removing a certificate | 18 |
| 3.6 | Error handling | 19 |
| 4 | Implementation | 20 |
| 4.1 | Application core | 20 |
| 4.2 | Empty State | 22 |
| 4.2.1 | EmptyState Component | 23 |
| 4.3 | Listing of certificates | 23 |
| 4.3.1 | General | 24 |
| 4.3.2 | Keys | 24 |
| 4.3.3 | Cert | 25 |
| 4.4 | CertificateActions | 26 |
| 4.5 | Requesting, importing and resubmitting a certificate | 26 |
| 4.5.1 | RequestCertificate | 27 |

| | | |
|----------|---|-----------|
| 4.5.2 | RequestCertificateModal | 27 |
| 4.5.3 | ResubmitCertificateModal | 30 |
| 4.5.4 | CAsRow | 32 |
| 4.5.5 | StorageRow | 32 |
| 4.5.6 | NicknameRow | 32 |
| 4.5.7 | CertFileRow | 32 |
| 4.5.8 | KeyFileRow | 32 |
| 4.5.9 | SetSigningParametersRow | 33 |
| 4.5.10 | SubjectNameRow | 33 |
| 4.5.11 | DNSNameRow | 33 |
| 4.5.12 | PrincipalNameRow | 33 |
| 4.6 | Removing a certificate | 33 |
| 4.7 | Error handling | 34 |
| 5 | Testing | 35 |
| 5.1 | Empty state | 35 |
| 5.2 | No certificates available | 36 |
| 5.3 | Warning of expired certificates | 36 |
| 5.4 | Modifying auto-renewal | 36 |
| 5.5 | NSSDB stored certificate | 37 |
| 5.6 | File stored certificate | 37 |
| 5.7 | Updating after receiving D-Bus signal | 37 |
| 5.8 | Removing a certificate | 37 |
| 5.9 | Importing a certificate | 38 |
| 5.10 | Requesting a certificate | 39 |
| 6 | Conclusion | 40 |
| | Bibliography | 41 |

Chapter 1

Introduction

Certificates play a crucial role in today's IT industry. They are an important part of SSL/TLS¹ and by extension security on internet. They are there to identify parties participating in secure communication and to prevent impersonation of any party by an impostor. Yet the way how they are requested and managed is often hard to grasp by administrators who are not proficient in this topic or users who are not even administrators. A command-line tool used to generate certificate request may be too confusing for many users and therefore a clear and simple user interface is needed. A result of this work should be an application that would allow a user to request and manage certificates with a smart user interface. This work would be part of a bigger project called Cockpit², which provides a user interface for managing Linux servers. The final application would be a part of the bigger Red Hat's portfolio. Therefore the main aim is to fit with Red Hat's³ existing identity management system FreeIPA⁴.

The following sections are going to explain the basics of security certificates and problems of modern Linux administration. Most of the information there comes from Red Hat documentation[11] and Symeon Xenitellis book The Open-source PKI Book[14].

1.1 Public key infrastructure

When establishing a secure connection it's important to authenticate that the other side of communication is the entity we mean to send data to. This authentication tries to resemble real-life situations when we can identify a person by their face, physical characteristics, or signature to establish communication with correct entity. In the digital age, it's equally important to identify that transfer was established with the correct entity in order to prevent a leak of sensitive data into an incorrect place.

Public key infrastructure is a system for an identification of entities in secure communication by digital certificates. It is used in such areas as emails, SSL/TLS protocols (which is the basis for HTTPS), electrical signatures, smart cards, etc.

Prevention of eavesdropping Public key cryptography, also known as asymmetric cryptography is a method for encrypting information before sending it and decrypting

¹Protocols for encrypting data to provide communication security. Widely used for internet, e-mails, messaging, etc.

²<https://cockpit-project.org/>

³A software company specializing in providing an open-source Linux oriented solutions

⁴https://www.freeipa.org/page/Main_Page

information after receiving it to prevent reading of the data during transfer. It is used in SSL/TLS protocols. It uses two keys to accomplish this during the process:

- Public key is key used to encrypt data. It can be shared publicly without danger of compromising data.
- Private key is key used to decrypt data. It is important to keep this key hidden to prevent compromising it.

This ensures that only intended receiver, the owner of the private key, can read encrypted data and nobody else has read the data during the transfer.

Prevention of tampering Another problem that needs to be addressed is to ensure that received information is information which the sender sent and have not been replaced by an intruder with their own. This is solved by digital signatures using the following method: A one-way hash of data is created which is then encrypted by a private key and sent. To validate the integrity of data the receiver has to decrypt the hash using the public key and then create their own hash from received data. If these two hashes match, it means data has not been tampered with since it was signed.

Prevention of impersonation An impersonation is an act during which a party pretends to be the intended receiver of information and supplies the sender with its own public key. Public key cryptography uses certificates to confirm the other party is who they claim to be.

1.1.1 Certificates

A certificate is an electronic document that provides proof of identity and confirms the public key. While parties can establish a secure connection with their private keys, they need to verify that they are communicating with intended entities. To ensure this, each party presents a certificate of identity which was signed by a certificate authority (CA) which party trusts. There are many standards for defining a format of certificates. In this work, we will talk about X.509⁵ certificates.

Certificate authority

A certificate authority is an organization that issues digital certificates. Certificate authority identifies an individual by name, such as organization name, person's name, or server's hostname. This name is then bound to a public key. This ensures that ownership of the public key can be verified. Furthermore, a certificate authority also inserts its own digital signature into the certificate which makes sure that certificate was truly signed by that certificate authority and can be valid for users who trust the certificate authority.

Issuing a certificate

Issuing a certificate is a non-unified process that differs from one certificate authority. Some certificate authorities may only require email address and password, others may require a background check or a personal interview.

⁵a standard for a format of certificates

To obtain a certificate, an applicant has to create a certificate signing request (CSR). The most common format for this request is PKCS 10, which describes a syntax for certification requests and its attributes. According to this standard, certificate signing request may contain information described in table 1.1.

| <i>Information</i> | <i>Description</i> | <i>Example</i> |
|--|---|--|
| Common Name (CN) | This is the fully qualified domain name (FQDN) of the device to be secured. | www.example.com *.example.com mail.example.com |
| Business Name/Organization (O) | The legal incorporated name of the organization. The name shouldn't be abbreviated, and it should include suffixes like .Ltd, .Inc. | AppViewX, Inc. |
| Department Name/Organizational Unit (OU) | The department in your organization handling the certificate. | IT, Finance |
| City/Locality (L) | The city/town your organization is located in. | New York City |
| Province, Region, County, or State (S) | This should not be abbreviated | New York |
| Country (C) | The two-letter ISO code of your country | US |
| Email Address (MAIL) | The primary point of contact in your organization for certificate-related operations, usually the IT department | helpdesk@example.com |

Table 1.1: Distinguished names[3]

The process of requesting and issuing a certificate is the following:

1. The applicant generates a key pair consisting of a private key and public key. They keep the private key secret.
2. The applicant generates a certificate signing request containing information identifying the applicant. It also contains the public key generated in the previous step.
3. The certificate signing request is sent to the certificate authority, which sends back a certificate signed by the private key of this certificate authority.

Tracking and renewing a certificate

Each certificate is issued only for a certain period of validity. It's important to keep track of issued certificates to know if a certificate is about to expire, so they could renew it. Renewal of certificate is a process with the same steps as the initial requesting of a certificate, where a signing request must be created, submitted to the certificate authority which then issues a new certificate. This certificate then replaces the old one. The process of renewing however reuses the existing key pair. The renewed certificate also keeps the properties of the old certificate, with the difference of new expiration. This allows the user to smoothly transition from the old certificate to the new one.

Alternatively, the applicant may want to revoke certificate. This may be a desired action in a situation such as when identifying properties for a certificate are no longer current or if the private key associated with the certificate has been compromised. This can be done by adding a certificate into Certificate Revocation List (CRL)⁶.

1.2 Cockpit and problems of Linux administration

„Cockpit is an open-source web-based solution for administering Linux servers“ [2]. As a project, it aims to make Linux administration easier, more accessible, and beginner-friendly. It mainly serves as a smart user interface to manage server resources. It aims to provide help to beginner system administrators or system administrators who are not native to a Linux environment. They might be Windows administrators who just need to do few tasks in Linux or administrators who just decided to switch from Windows to Linux. This target market is usually not very familiar or uncomfortable with the Linux command line world and they might prefer simple intuitive UI. Cockpit tries to make it's UI understandable without a need to read manual pages or documentation. The UI should ideally navigate the user to achieve the desired outcome. Cockpit aim is to show the real-time state of a resource and it to apply server changes immediately when the user does so through user interface. Therefore Cockpit doesn't save any configuration on the system nor does it impose any predefined state on the server. Each field of its administration is defined in encompassed in an insertable plugin, which acts as an independent part of the software and can be developed independently even outside of the cockpit project itself. There are plugins for administering various resources in various fields such as storage, networking, virtual machines, containers. It's common for each plugin to use one main technology for administering a type of resource and therefore act as a UI for this preexisting backend. For example, the containers plugin mainly uses Podman, the virtual machines plugin uses the Libvirt virtualization library. A Result of this work, a plugin for administering certificates, will be developed in the same manner and will act as a UI mainly for preexisting library Certmonger.

Cockpit also provides various D-BUS and file APIs which are used. They can be found at Cockpit's documentation ⁷ It also provides its own implementations of some design components.

⁶A list of certificates which has been revoked by their certificate authority

⁷<https://cockpit-project.org/guide/latest/>

Chapter 2

Technologies

This chapter describes various languages, libraries, and frameworks used for the development of frontend, communication with backend, and testing.

2.1 Javascript

„JavaScript is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web page“ [7]. It’s dynamically typed and based on standard ECMAScript. Mainly used for the development of a website’s client-side part. It is now widely supported by common web browsers.

Javascript has a wide variety of libraries popular for web development, such as React.

2.2 React

„React is a Javascript open-source library meant for building user interfaces“ [10]. Created in 2013 and developed by Facebook and the open-source community it is now one of the most popular JavaScript libraries for building web-based user interfaces. React is component-based. It means that each part of UI can be encapsulated into a component and broken down into many other smaller components, creating a hierarchy, with a top component added into a DOM of a website using:

```
ReactDOM.render(App, document.getElementById('root'));
```

Where `App` is the top component of the hierarchy which renders the whole React application.

Each component is able to manage its own state and re-render parts of UI when the state changes. It also has the ability to contain the markup part and logic part of UI together in the same file.

2.2.1 Javascript XML

Javascript XML, or JSX, is a syntax extension of Javascript introduced as part of React. Thanks to JSX, React is able to present both markup and rendering logic together in the same component. It might look like a mix between Javascript and HTML, there are few differences and unique things when compared to each respectively. JSX allows you to embed an expressions inside it’s markup elements:

```
const element = (<h1>My favorite number is {2 + 2}</h1>);
```

Similarly you could use variables or function calls in middle of markup.

It also allows you to use conditions for rendering certain elements in markup. For example:

```
let element;
if (showGreeting)
  element = <h1>Hello!</h1>;
```

Can be written as:

```
const element = showGreeting && <h1>Hello!</h1>;
```

Similary you can use ternary condition when rendering elements in if-else manner. Instead of writing:

```
let element;
if (isTextBold())
  element = <b>{renderedText}</b>;
else
  element = <span>{renderedText}</span>;
```

Can be written as:

```
const element = isTextBold() ? (<b>Text</b>)} : (<span>Text</span>);
```

2.2.2 Stateless components

Stateless components are components that do not store data about themselves. They usually pass their data to a parent component or are used just to render read-only data without any expectation of receiving input from the user.

An example of a typical stateless component which only renders properties passed to it is:

```
const NameRow = ({ name, surname }) => {
  return (<
    <span>Your name is {name}</span>
    <span>Your surname is {surname}</span>
  </>);
};
```

2.2.3 Stateful components

Stateful components are on the other hand components which store data in its state store which operates as a JavaScript object. Thus the object remembers information about itself. These state stores are used to store information that is expected to change dynamically. For example, it can be used to store user inputs or other values which are the result of user actions. Whenever the state of a component changes, the component itself is re-rendered including all children components. A result of this is a reactive UI that dynamically reacts to user actions or to outside changes, like a signal from outside of the application, without a need to reload the page. The trigger of this change is when a function `setState` is called which sets new values of component's state store. A stateful component is implemented as an extension to JavaScript's `class`, which means it can have its own class properties

and methods. These components also have „life-cycle methods“ you can override such as `ComponentDidMount` which is called when the component is inserted into the tree and is good to be used for fetching data from asynchronous calls or to set up subscriptions.

Below is an example of a typical stateful component. It defines the initial content of state store in the constructor, defines a helper method, and returns a renderable content which re-renders every time state updates:

```
export class Name extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "",
      surname: "",
    };
    this.onValueChanged = this.onValueChanged.bind(this);
  }

  onValueChanged(key, value) {
    this.setState({ [key]: value });
  }

  render() {
    const { name, surname } = this.state;

    return (<>
      First name: <input type='text' value={name} onChange={e =>
        onValueChanged("name", e.target.value)} />
      Last name: <input type='text' value={name} onChange={e =>
        onValueChanged("surname", e.target.value)} />
      <span>Hello {name} {surname}!</span>
    </>);
  }
}
```

This example has, of course, some undesirable behavior from UX point of view, such as the greeting sentence will re-render as the user will type their name. For purpose of this demonstration, it is sufficient.

2.3 Patternfly

„Patternfly is an open-source design system created to enable consistency and usability across a wide range of applications and use cases.“^[8]

Its library exports a variety of design components (such as Button, Form, Accordion, etc.) and Layouts (such as Grid, Flex, Stack, etc.) in HTML/CSS or as React components.

The project is heavily supported by Red Hat and the open-source community. Patternfly, its components, and guidelines are used for a variety of Red Hat products in order to offer consistency among the design of its products.

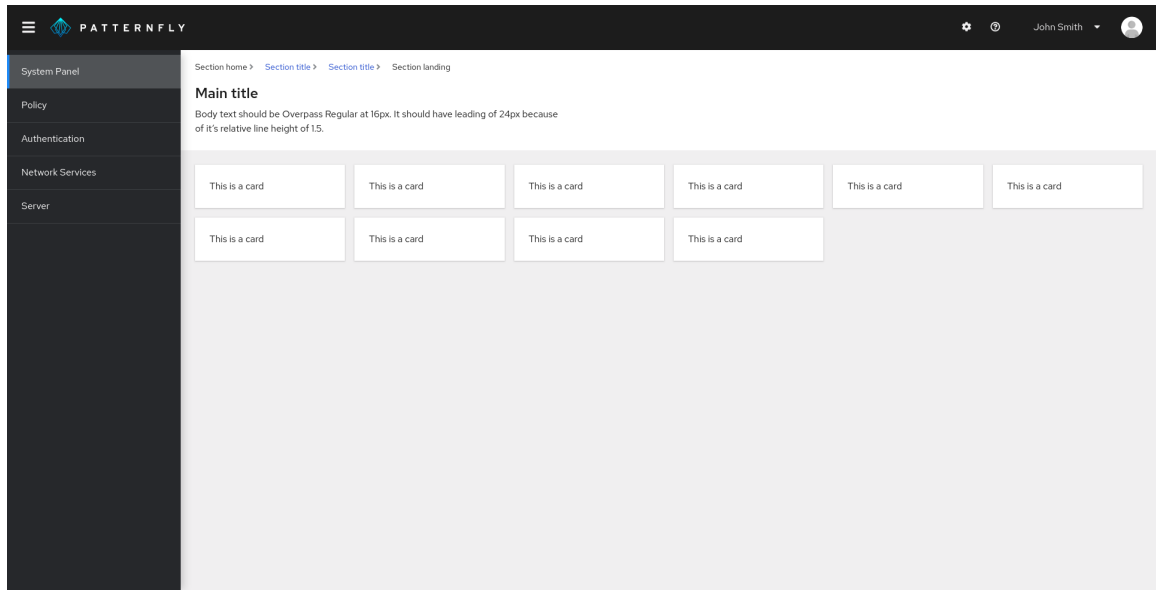


Figure 2.1: An example of page build with Patternfly

2.3.1 Fundamentals

- **Flexibility:** This means components are flexible to be arranged and customized in different ways. Patternfly offers space for adding custom CSS to its components.
- **Accessibility:** Patternfly makes sure its components are accessible for various groups of users. It makes sure the color scheme is suitable for color-blind users, it makes sure its components are labeled in order to be analyzed by screen readers used by blind users, buttons are large enough for users with hand-motor disabilities, etc.
- **Consistency:** helps to create unified applications and interfaces. All products in a company portfolio have unified design and users can switch between products without having to adapt to a different design and learn to use a new product from scratch.
- **Scalability:** This means its components are designed for business of all sizes.
- **Make it open:** It means Patternfly is developing its library according to open-source principles.

2.3.2 Designer Guidelines

Patternfly also comes with a series of designer guidelines. These guidelines are divided into 3 areas:

- **Styles:** This section explains matters like correct usage of colors for various states (like danger, success, warning), color contrast ratio, correct usage of icons, and standardization of fonts and spacing.
- **Usage and behavior:** This describes correct usage for various design components provided by Patternfly. It explains how to use components in certain situations, where they should be placed, when to use alternative components instead, etc.

- Content: This section touches on issues of correct usage of terminology (e.g. when to use the word 'Remove' and when to use the word 'Delete'), sets rules for using abbreviations, acronyms. It specifies in which situation to use sentence-style capitalization and in which situation to use title style capitalization. It recommends formats for date and time and many other things. It also talks about writing effective text according to UX principles.

2.4 D-Bus

„D-Bus is a message bus system, a simple way for applications to talk to one another“ [4]. It allows for inter-process communication on the same host between 2 applications with the possibility of routing between 1 and 0-N applications. The purpose of D-Bus is mainly to simplify communication between desktop applications or between desktop session and the operating system. It has multiple wrappers and implementations for other languages or frameworks such as Python, Qt, cockpit-dbus. These implementations are what most people use because it simplifies the usage of D-Bus. The specifics of D-Bus which will be described here might slightly differ from one D-Bus framework implementation to the other, but most of them should contain similar same basic principles:

2.4.1 Objects

D-Bus objects give the means to refer to instances and specify methods and signals which can be used on such instances. Objects are referred by object paths, which look like file system paths. An example of object path is `/org/fedorahosted/certmonger/requests/Request6` Objects also support interfaces, which represent a group of methods and signals. Each object also contains interface `org.freedesktop.DBus.Introspectable` with method `Introspect`. This is useful to find out available methods, interfaces, and signals of an object.

Object method

Object methods are instances that can be invoked on an object with optional inputs and outputs. To call a method, the call message is constructed containing bus name, name of the method, arguments, object path, and optionally the interface of that method. This message is then sent to the bus daemon, which is forwarded to the destination process or error message is returned. The destination process receives and evaluates the method and sends back a reply message to the bus daemon. The bus daemon forwards it to the process which invoked the method call.

Object signals

Signals are event-like messages which are broadcasted by the emitted (only if the user subscribes to them). To receive a signal, a receiver has to subscribe to the bus daemon by registering „match rules“ which tells which signals ts is interested in. When the event emitting signal happens, the emitter creates a special type of signal message consisting of the signal name, the bus name of the emitter process, and arguments. This message is sent to the bus daemon, which determines which processes it should be broadcasted to and forwards the signal.

2.4.2 Messages

Messages sent with D-Bus are formatted, consisting of header, body, and data payload. Routing of these messages is done through the bus daemon. Types of messages:

- Invocation of a method
- Result of an invoked method
- Exception in case of an error of an invoked method
- Signal method, which is emitted when an event has occurred

Details about D-Bus were taken from Freedesktop D-Bus documentation[5] and Python D-Bus documentation[6].

2.5 Certmonger

„Certmonger is a service that attempts to simplify interaction with certifying authorities (CAs) on networks that use public-key infrastructure (PKI).“[1] Certmonger simplifies this interaction by encompassing multiple steps of issuing, renewing, re-keying, resubmitting, and revoking a certificate in a single utility. Certmonger also offers the ability to track and automatically renew a certificate before expiration without the need for any user interaction. Taking into account steps described in subsection [Issuing a certificate](#) and need to track a certificate described in subsection [Tracking and renewing a certificate](#), certmonger simplifies this process and helps with the problematic by offering the functionality:

- It generates a certificate signing request from identifying information supplied by the user. It can also generate a key-pair as part of a certificate requesting process or allow the user to supply their own key-pair.
- It can submit the generated request to a CA, wait for CA to issue a certificate, and store it.
- It can track a certificate, notify a system administrator when a certificate is about to expire, and automatically renew a certificate before it expires.
- It can resubmit with new identifying properties, in case they need to be amended, or if user wishes to renew a certificate manually.
- It can manage a list of certificate authorities it can communicate with, allowing the user to add new certificate authorities if they provide needed configuration or SCEP¹ interface.

It also provides command-line tool `getcrt` which served as inspiration for this application. Certmonger also offers D-BUS APIs so its functionality can be exploited by other applications. Table 2.1 contains a list of APIs offered by certmonger relevant to this work's implementation:

¹A protocol for enrolling a certificate using URL

| <i>Interface</i> | <i>Method name</i> | <i>Description</i> |
|-------------------------------------|---------------------------------------|---|
| org.fedorahosted.certmonger | <code>add_request</code> | Requests or imports a certificate |
| | <code>find_request_by_nickname</code> | Finds certificate request by its nickname |
| | <code>get_requests</code> | Returns a list of D-BUS paths of certificate requests tracked by certmonger |
| | <code>remove_request</code> | Stops tracking a certificate request. This however does not revoke it. |
| org.fedorahosted.certmonger.request | <code>modify</code> | Modify properties of the certificate request. Changes will come into effect with next resubmitting. |
| | <code>rekey</code> | Generate a new key-pair |
| | <code>resubmit</code> | Resubmit request to the CA using the same key-pair |

Table 2.1: Certmonger APIs

Chapter 3

Design

This section described problems, use cases, and proposed designs for each functionality.

A Patternfly's guidelines, which were described in section 2.3.2 were taken into account while developing a design in order to provide a unified approach which would fit with other Red Hat's user interface solutions.

The main aim here is to design a UI, which would be easy to understand for beginner administrator. It would not overwhelm users with overcrowded forms. It would try to navigate the user to get a result that would satisfy most users, in case the user is not familiar with technical details concerning a topic of certificates. It would also try to be faster to go through as existing command-line solutions.

It tries to address problems of a unified approach to error handling, and provide designs for functionalities of requesting, resubmitting, importing, and removing a certificate.

Several similar solutions are available, such as certmonger's command-line utility `getcert`. This design was created after reviewing an existing command-line utility `getcert` provided by certmonger was used many times as inspiration while designing UI to find out what parameters are important for each action and what behavior is expected in certain situations.

3.1 Listing certificates

If users have a certificate enrolled in their system, it's important for them to track it. A certificate can exist in a state of just being a file somewhere on a system, but in this scenario, the user cannot easily get information about the certificate itself. Some information can be critical, such as the expiration date, so the administrator doesn't miss the expiration date.

The following design is proposed: Have a list of certificates available. Each certificate will have a header with these values:

- Identifying value of a certificate: In the case of NSSDB stored certificate, it's a nickname. In the case of file stored certificate, it's an ID of a certificate.
- Certificate's expiration date. If the certificate expires the next day, that day or has expired the previous day, use strings „tomorrow“, „today“ or „yesterday“. It should also show if it's going to automatically renew or expire. Examples: „Expires today at 14:15“, „Auto-renews before 21/06/2021“ If it's going to expire soon, meaning in less than 28 days, show a warning icon. If it already expired, show an error icon.
- Certificate authority which issued the certificate.

- A drop-down available with a certificate's actions. These actions are resubmitting a certificate and removing a certificate.

Users can introspect the certificate by clicking on it. Here more properties of the said certificate can be listed. If a property can be amended, such as automatic-renewal, it should be allowed for the user to update. Certmonger tracks dozens of properties for each certificate which are available through its D-BUS interface. The full list can be found at Pagure website¹. Rendering all of these properties would crowd the page. Therefore only a few properties were chosen to be rendered, which can be divided into multiple categories. Each category should be shown separately from each other in the UI, ideally by tabs:

- **Properties of a certificate request itself:**

- Status - A certificate request's status. If certificate is stuck, a warning icon is shown next to it signaling that request got stuck at certain undesirable state.
- Validity - Here is shown a whole validity range by combining `not-valid-before` and `not-valid-after` values. An example: „10/06/2020 to 12/05/2021“.
- CA - Again a certificate authority which issued a certificate is shown here.
- Auto-renewal - Shows if certificate will be automatically renewed before it expires. User can change this property here. Therefore a checkbox should be present where user can select if they want automatic renewal or not.
- Subject name - A name of a subject which is being secured. It can consist of values such as common name, organization name, etc.
- DNS names - A list of DNS names which are part of subject alternative names.
- Principal name - A name used for Kerberos².

- **Private key properties:**

- Nickname - Nickname used to identify a private key in NSSDB.
- Type - Type of algorithm used to create a private key.
- Token - NSSDB token.
- Storage - Storage type used for this key. Can be either „NSSDB“ or „File“.
- Location - Location of private key's file or NSSDB where it's stored.

- **Certificate properties:**

- Nickname - Nickname used to identify a certificate in NSSDB.
- Storage - Storage type used for this key. Can be either „NSSDB“ or „File“.
- Location - Location of a certificate file or NSSDB where it's stored.

This list made by looking which properties are exposed by command:

```
#getcert list
```

.

¹<https://pagure.io/certmonger/blob/master/f/src/tdbus.h>

²cryptology authentication protocol

3.2 Requesting a certificate

Probably the most important action is requesting a certificate itself. Let's have a use case where a system administrator administers a server with a website of a bank. This server is also enrolled with FreeIPA³. It's vitally important that users trying to load this website are connected to the right party, and not somebody who is trying to impersonate the bank website. (an act of impersonation is described in section 1.1).

Usually, to request a certificate, an administrator would use some tool, such as OpenSSL⁴. They would have to go through the process of generating a private key. In most cases, this can be done automatically, as most users are satisfied with the 2048-bits RSA key. Then they would have to generate a signing request where they need to specify signing request properties such as common name and subject alternative name. It might be confusing to the administrator which of these parameters are compulsory and which are optional. Some parameters, like common name, can be generated automatically from the server's hostname. Next, they have to submit the request to a certificate authority, obtain the certificate, and store it somewhere.

Certmonger deals with the problem of submitting a request to a certificate authority, obtaining it and storing it. However, it's command-line interface offers a lot of options with a lot of parameters and sometimes it's not clear which parameters are to be used. This might be discouraging for a beginner administrator or administrator who's primarily working with Linux.

To solve the above-mentioned problems, the following design is proposed: Have a way, ideally a „Request Certificate“ button, visible at all times and easy to find. It would be placed on the top right corner above the certificate listing. This place would create an „action area“ of some sort. The button would open a dialog with a form for requesting a certificate. The form would allow the user to choose the following inputs:

- Certificate authority: User can choose from a list of certificate authorities pre-configured by certmonger.
- Storage type: User can choose if they want to store a certificate in NSSDB (default option since it requires the least amount of configuration)
- Additional signing request parameters like common name, DNS name, and principal name. Make visible that these parameters are optional. These 3 parameters were chosen out of many others because they occur the most at documentation and user issues related to certificates at Red Hat Customer Portal⁵.

Immediately after requesting a certificate, leave the dialog open to wait for either a success or an error. In case of an error, leave the dialog open and print an error message at the foot of the dialog. In case of success, close the dialog. At this point, a successfully added certificate should in the list of certificates.

Also, pre-generate inputs with default values, if possible, for the user in order to navigate this functionality with the least amount of time required.

³Open-source identity management system

⁴OpenSSL is a cryptography library that can be used to generate private keys and certificate signing requests

⁵<https://access.redhat.com/>

3.3 Importing a certificate

Let's have a case where the user already has obtained a certificate from certificate authority beforehand and has a certificate and key-pair stored on their system either in the NSS database or in files. They have the certificate itself but they do not track it nor they have an automatic renewal setup. Therefore it should be possible to allow them to add their own certificate into the system so they could have a way to track it, renew it, or set up automatic renewal. This process should be very similar to that of requesting a certificate described in 3.2. The difference is that the first steps of requesting a certificate: creating a private key, creating a signing request, submitting a request are skipped, as in this case user has already done it beforehand and we go straight to storing, tracking, and managing an imported certificate.

Therefore the following design is proposed: Place an „Import Certificate“ button next to a button for requesting a certificate. Together these buttons make up an „action area“ described in 3.2. Clicking this button would open a dialog. This dialog again would be almost identical to one for requesting a certificate and would differ only in the following points:

- Default storage type would be „File“. After studying documentation and issues at Red Hat Customer Portal⁶ it seems that when user import certificate, in most cases they have a certificate stored in a file. This is in contrast to when users request a new certificate, they mostly store it in NSSDB. That's why the default storage option should be „File“.
- Paths for the certificate file and key-pair file must always lead to an existing file on file system. This is again in contrast to requesting a certificate where the user can specify a location of a non-existent file as certmonger will later create a new certificate at this location.
- User cannot specify signing request parameters such as common name, DNS names, and principal name. When importing a certificate, the process of creating and submitting a signing request has already been done and these properties are already part of certificate which user is trying to import.

The similarity between requesting a certificate and importing a certificate could raise the question: Why split this into two modals? Alternatively, we could have one button which opens one dialog where the user could select a „mode“ if they want to import an existing certificate or request a new certificate. However one of the main things which is trying to be avoided is to not make a dialog for requesting a certificate too large and overfilled with various inputs that would overwhelm a user when they open the dialog. That's why from a UI standpoint it's better to have two separate modals for these functionalities even though they may share most of the code.

3.4 Resubmitting a certificate

Resubmitting a certificate is an action where the user wants to renew an existing certificate or amend some properties of an existing certificate. They already have enrolled certificate

⁶<https://access.redhat.com/>

with existing key-pair and defined signing request parameters such as subject name, principal name, and DNS names. In this case, they would like to resubmit the same certificate request as last time. This is a use case on many occasions. For example, if the user does not have automatic renewal set up and wants to renew such certificate manually. Technically there is no such action as certificate renewal. If one person wants to renew a certificate, they have to generate a new signing request using the parameters and key-pair of the old one, submit it to a certificate authority and then replace the old certificate with the new one. Such procedure can be done by resubmitting a certificate. Another case when resubmitting a certificate comes in handy is when the user wants to update some parameters of a certificate. For example, they want to add or remove some domains from list of DNS names, but leave everything else intact. In that case, it's useful to allow editing some parameters when resubmitting a certificate.

So in order to comply with above-mentioned problems, the following design was proposed: Each certificate will have a button present to resubmit it. This button will open a dialog that will allow to amend some properties of the certificate. Properties such as storage, which cannot be amended will be shown as read-only information in the dialog. Then a certificate authority, subject name, dns names and principal name will be filled with values from the old certificate. It will be available for editing, but ideally, users might skip doing any changes and just click on „Resubmit“ button without doing any additional changes if they do not need to.

3.5 Removing a certificate

If a user wants to get remove a certificate, they can do multiple things. They can just merely tell certmonger to stop tracking. Certmonger then doesn't keep the certificate in its list of certificate and doesn't automatically renew it. However if users have a certificate and private key stored in files somewhere on their system, this doesn't remove those files. Usually to do full cleanup users have to delete those files too. On the other hand there are use cases when they would like to keep those files, for example if they want to move them or import them again into certmonger with a different configuration. All of this functionality can be implemented into this application. There are other things users should do when they delete a certificate. If the said certificate is used by some other application, let's say an Apache server, they would probably want to update the configuration of that server so it won't try to use a certificate which was removed. Also if the removed certificate was compromised, users would want to add said certificate into a certificate revocation list. These actions however cannot be presented in the UI, since the application doesn't have a way to know which processes use a certificate which user wants to remove. Also adding a certificate into a certificate revocation list is a process that is unique for each certificate authority and thus this process goes beyond the capabilities of this application.

After explaining problems and what options are available, the following design is proposed:

Each certificate will have a button present to remove it. Since removing a certificate can be a very destructive operation, this button will open a confirmation dialog. This dialog will render the certificate's identifying properties. In case of a certificate stored in the NSS database, this will be a path to the NSS database itself and a nickname used to identify a certificate in this database. In case of a certificate stored in a file, the identifying property is a location of this certificate's file. Showing a location of the certificate's private key is also welcomed. Furthermore, in case of a certificate stored in a file, there will be an

option, ideally, a checkbox, which will allow a user to choose if they want to also remove a certificate file and certificate's private key file.

3.6 Error handling

An important part of the application was a unified approach to handling errors. The design here tries to solve 2 problems: where to display errors and how to display multiple errors at once. The first problem offers multiple approaches. An error can be displayed „on spot“ inside the component which caused the error. That means that if an error for example was caused by some action from a modal, it should be displayed in the modal. This however would cause problems if the user is no longer focused on that part of UI. Most of the backend calls are asynchronous, so it's possible that backend call that can cause error can take several seconds to evaluate. The user in meantime could no longer be at that part of UI.

The other approach is to show errors at an application-wide shared spot. This could be some side panel or pop-up. The benefit of this pop-up is that it's possible to stack multiple errors on top of each other if the situation demands. In this implementation, I decided on a combination of both approaches. Places like modals are not closed until asynchronous action is fully evaluated, thus it's impossible for a user to leave modal until we are sure there was no exception. In every other place, a pop-up is used.

Chapter 4

Implementation

The implementation of this application derives from Cockpit's starter-kit¹, which serves as a template for Cockpit's plugins. Most of the implementation was done in React (section 2.2) using components provided by Patternfly (section 2.3). As such, whole application can be systematically divided into React components. Most of components usually try to implement topics described in design section 3.

Therefore it's convenient to explain the implementation of each design topic by explaining its React components.

4.1 Application core

The `Application` component, which can be found in `src/app.jsx`, is the core of the application. It's directly inserted into a DOM and it's the parent component for all the other components which are part of the application. At this place a lot of logic, which can be most comfortably done at the uppermost level of the application, is taking place. Here can be found logic for managing a list of certificates, certificate authorities, rendering application-wide error messages, subscribing to and monitoring the state of certmonger service. Each of these processes is explained here:

Monitoring certmonger service - When `Application` component is mounted, a method `subscribeToSystemd` is called which subscribes the application to systemd client. The process of how subscribing to dbus client works is described in 2.4.1. This then listens to any changes to certmonger service and updates a state of the service which application tracks. A change to this service can trigger `EmptyState` component described in section 4.2 to allow to troubleshoot inactive or dead certmonger service. There is also method `this.updateCertmongerService` which tries to start certmonger service if it's not active. However, it tries it exactly once, when the application mounts and the service is not active at the moment. It does it only once to prevent cyclical booting of certmonger service or to prevent automatic booting of the service in case user tries to deliberately kill it.

Monitoring certificates and certificate authorities - Similar to the previous paragraph, a method `subscribeToCertmonger` is called which subscribes the application to a certmonger client. This then listens to any signals which could mean a change in a list of certificates or certificate authorities. If a change happens, a list of certificates or certificate authorities kept in the application's state is updated accordingly.

¹<https://github.com/cockpit-project/starter-kit/>

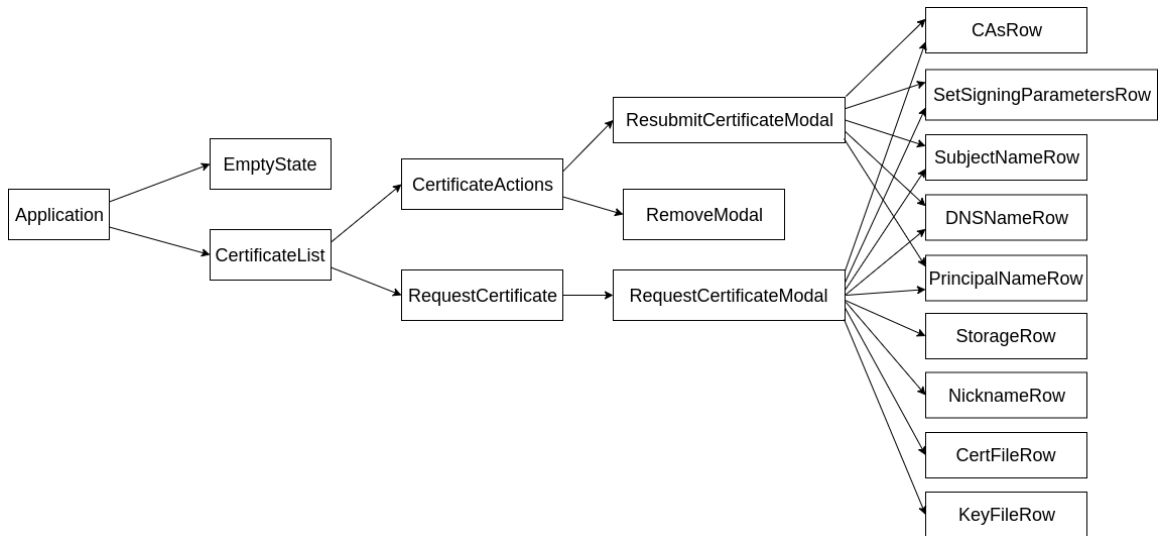


Figure 4.1: A project structure consisting of React components

Setting warning in Cockpit's navigation - While all certificates are fetched, their expiration date is also checked with method `checkExpiration`. If there is any expired certificate, an error is set using Cockpit's `page_status` API, which shows in a form of error icon next to page's item in Cockpit's menu. Similarly, if a certificate is close to expiration, a warning is set.

Keeping a list of certificates and certificate authorities - When components mounts, methods `getCertificates` and `getCertificateAuthorities` are called which load these lists and save them into application's state. These lists are kept in this component, at the top level of the application so they could be passed down to any child component which might need them or updated by a callback.

Managing application-wide error messages - This component also keeps a list of error messages.

Its state has these properties:

- `alerts` - a list of error messages.
- `certmongerService` - an object which keeps properties of certmonger service, so service's state can be checked at any moment. It is set by `updateCertmongerService` method.
- `initialPhase` - a boolean flag initially set to true, which is used to ensure that inactive certmonger service is tried to start only once - after mounting the application. It is then set to false and the service is not tried to start anymore.
- `cas` - a list of certificate authorities. New certificate authorities are added by `getCertificateAuthority` method.
- `certs` - a list of certificates. New certificate are added by `getCertificate` method.
- `expiredCerts` - a number that keeps a track of how many certificates have already expired. Set by `checkExpiration` method.

- `toExpireCerts` - a number that keeps a track of how many certificates are going to expire soon. Set by `checkExpiration` method.

It consists of several methods:

- `onValueChanged` - a helper method used to update the component's state. It's used as a callback which is passed down to `CertificateList` component as `appOnValueChanged` parameter. This way application's state properties, such as list of certificates, can be updated from some child component.
- `componentDidMount` - React method which is called when component mounts. Here it's used to call methods `subscribeToCertmonger`, `subscribeToSystemd`, `updateCertmongerService`, `getCertificateAuthorities` and `getCertificates`.
- `subscribeToSystemd` - Method which subscribes over a `Systemd` client `org.freedesktop.systemd1` using Cockpit's `cockpit.dbus.subscribe()` API. It listen signals which match an object path `/org/freedesktop/systemd1/unit/certmonger_2eservice` on `org.freedesktop.DBus.Properties` interface. This way, every time a state of certmonger service changes, the `updateCertmongerService` method is triggered and the application is updated with a new state of certmonger service.
- `updateCertmongerService` - Method which fetches and updates a state of certmonger service using Cockpit's `service.proxy` method. If the application is in initial phase and service is stopped, it tries to start it.
- `subscribeToCertmonger` - Method which subscribes over a certmonger client `org.fedorahosted.certmonger` using Cockpit's `cockpit.dbus.subscribe()` API. It listen any signals on `org.freedesktop.DBus.Properties` interface, which then triggers a callback. This handler checks if interfaces for certificates or certificate authorities, `org.fedorahosted.certmonger.request` and `org.fedorahosted.certmonger.ca` respectively, triggered these signals. If it updates properties of these objects which has changed, or adds an entirely new object in case a new certificate or certificate authority was added using methods `getCertificate` and `getCertificateAuthority`.
- `getCertificates` - Calls a function `getRequests` to get a list of certificate requests D-BUS paths and for each request calls a method `getCertificate`
- `getCertificateAuthorities` - Calls a function `getCAs` to get a list of certificate authority D-BUS paths for each authority calls a method `getCA`
- `getCertificate` - Calls a function `getRequest` to get an object of certificate request and updates application's state with this object.
- `getCertificateAuthority` - Calls a function `getCA` to get an object of certificate authority and updates application's state with this object.

4.2 Empty State

Before even the body of the application that handles certificate related functionality is even rendered, it's important to check if `certmonger.service` is present and running. In

case it's not present, running or there is any other problem related to this service which would disable the functionality needed to manage certificates, the `EmptyState` component is rendered.

4.2.1 EmptyState Component

This component is contained in `src/emptyState.jsx`. It uses cockpit's general implementation of empty state from `lib/cockpit-components-empty-state.jsx`, which in turn is wrapper of Patternfly's React [EmptyState component](#).

This component can be in 3 states according to the state of `certmonger.service`:

- Loading the certificate service: This is usually shown in a few seconds long time window when the application is still subscribing to `systemd` to monitor
- Starting the certificate service: This state is shown in a small time window when `systemd` is only starting the service. `certmonger.service` or fetching information about the service, therefore application doesn't have any information its state.
- The certificate service is not active: This is shown when the state of service is other than „running“ or „starting“. It usually means the state of this unit according to `systemd` is inactive (it may mean that it crashed or was stopped). In this case, 2 action button to start the service is provided. One start the service and the other offers the user to troubleshoot the problem by redirecting them to cockpit's services page where they can inspect `systemd` unit logs and do some further actions.



The certificate service is not active

Start the certificate service

[Troubleshoot](#)

Figure 4.2: A of `EmptyState` component

4.3 Listing of certificates

This is the part of the application that can be considered a homepage. It greets a user when they enter the application (if they do not have any trouble with service) and serves as a bridge to all the other functionalities related to management of certificates.

It is implemented in `src/certificateList.jsx` as a stateful component `CertificateList`. It uses cockpit's wrapper component for listing multiple expandable rows - `ListingTable`. It's mainly used to list resources, in this case, certificates.

Each certificate row has a header which shows identifying and most important properties, such as certificate nickname, validity period and a certificate authority which issued the certificate.

An important function of this section is semantics done for rendering an expiration time for each certificate. It's important to show the difference between an expired certificate, which might be a security vulnerability, a certificate that is going to expire soon, which would need a system administrator's attention and a certificate that has an automatic renewal setup or has a long time until its expiration.

To show an expired certificate, a combination of a red-colored Font Awesome Icon [fa-exclamation-circle](#) and date, when certificate expired, is shown.

If the certificate is going to expire soon, meaning no automatic renewal is setup and the certificate expires in less than 28 days, a combination of a yellow-colored Font Awesome Icon [fa-exclamation-triangle](#) and date, when certificate will expire, is shown.

The value of 28 days is derived from certmonger's 28 days policy „The certmonger service automatically renews the following certificates 28 days before their expiration date:

- CA certificate issued by the IdM CA as the root CA
- Subsystem and server certificates issued by the integrated IdM CA that are used by internal IdM services“^[12]

After expanding the row, 3 tabs are present: General, Keys, and Cert

4.3.1 General

Explains general information about the certificate request itself. The properties shown here are:

- Status: Shows certificate status in form of a state name and tooltip explaining details about the state. During the process of requesting or resubmitting a certificate, the request goes through multiple states. The full list of these states can be found at [manpages](#)² and a state logic found at [Pagure website](#)³. It's possible for a request to get stuck at some stage during the process. In that case, a warning icon is shown with a state at which it got stuck.
- Validity: This field combines certificate's `not-valid-before` and `not-valid-after`. Formatted using <https://momentjs.com/docs//parsing/string-format/> library to give a nice string of `not-valid-before to not-valid-after`
- Auto-renewal: Shown in a form of checkbox. Users can customize here if the certificate request will be automatically renewed 28 days before expiration.
- Signing request properties: These properties consist of Subject name, Principal name and DNS name.

4.3.2 Keys

Explains details about a key of the certificate. If a certificate is stored in a NSS database, properties here are mostly identical with properties in a Cert tab.

²<https://manpages.debian.org/stretch/certmonger/getcert-list.1.en.html>

³<https://pagure.io/certmonger/blob/master/f/doc/design.txt>

- **Nickname:** This property is present only if NSS database is used for storage to identify the key in a database.
- **Type:** Type of encryption of the generated key-pair. The default value of RSA⁴ with DSA⁵ and EC also available.
- **Token:** Optional token used only for NSS database.
- **Storage:** Specifies storage type. Possible values are only NSSDB or FILE.
- **Location:** Specifies the location of key-pair. In case of NSSDB storage type, this specifies a path to the database itself. In case of FILE storage type it specifies a path on file system to key-pair file.

4.3.3 Cert

Explains details about a certificate file of the certificate request. If certificate is stored in a NSS database, properties here are mostly identical with properties in a Key tab.

- **Nickname:** This property is present only if NSS database is used for storage to identify the certificate in a database.
- **Token:** Optional token used only for NSS database.
- **Storage:** Specifies storage type. Possible values are only NSSDB or FILE.
- **Location:** Specifies the location of a certificate. In case of NSSDB storage type, this specifies a path to the database itself. In case of FILE storage type it specifies a path on file system to certificate file.

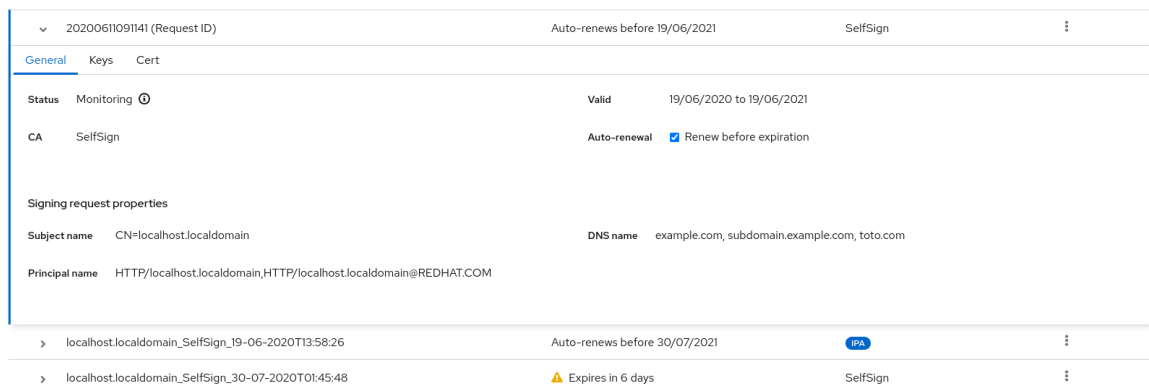


Figure 4.3: A screenshot of a list of certificates

⁴Rivest–Shamir–Adleman algorithms used for public-key encryption

⁵Digital Signature Algorithm

4.4 CertificateActions

This component handles representing actions which can be done for each certificate. It's done in a form of drop down menu presented by three dots called in UI terminology as „kebab menu“. It's present as fourth and last column in header of each certificate and therefore is present without need to open a tab with certificate itself. Two actions are present inside of this drop down: certificate removal and certificate resubmitting. It also manages which dialog is opened for each of this action.

It uses Patternfly's [Dropdown](#), [DropdownItem](#) and [KebabToggle](#) components.

Its state has these properties:

- `dropdownOpen` - a boolean flag which says if dropdown of options is open. This is needed because of the nature of Patternfly's `Dropdown` and `KebabToggle` which do remember the state of drop down is open or not so it has to be kept externally by those who use this component.
- `showRemoveModal` - a boolean flag which says if dialog for removing a certificate is open.
- `showResubmitModal` - a boolean flag which says if dialog for resubmitting a certificate is open.

The class of this component has several methods:

- `onValueChanged` - a helper method used to update the component's state. It's mainly used to clear a callback for changing a state which can be passed to Patternfly's `Dropdown` and `KebabToggle` components.
- `onResubmitModalOpen` - a helper method used to open dialog for resubmitting a certificate. It's passed as `onClick` callback to `DropdownItem`.
- `onResubmitModalClose` - a helper method used to close a dialog for resubmitting a certificate. It's passed as parameter to `ResubmitCertificateModal` so user can close the dialog from inside.
- `onRemoveModalOpen` - a helper method used to open dialog for removing a certificate. It's passed as `onClick` callback to `DropdownItem`.
- `onRemoveModalClose` - a helper method used to close a dialog for removing a certificate. It's passed as parameter to `RemoveModal` so user can close the dialog from inside.

4.5 Requesting, importing and resubmitting a certificate

This is probably the most important action of whole application. User can reach this functionality by finding „Request Certificate“ and „Resubmit Certificate“ buttons in a top right corner above the list of certificates.

It is implemented in `src/requestCertificate.jsx` as series of stateful and stateless components.

4.5.1 RequestCertificate

This component deals with a logic of rendering a buttons for opening a modal to request or import a certificate. It uses Patternfly's [Button](#) component. The component works in two modes: import or request. Depending on value of `mode` prop it renders a button with different text and opens `RequestCertificateModal` in either mode. It has 2 class methods: `onOpen` and `onClose` which change state property `showDialog` of the component. This property determines if the dialog to request/import a certificate is rendered or not. It might not allow opening of the dialog if it determines no certificate authorities are available.

`onOpen` method sets `showDialog` as true and is passed as handler to `onClick` argument of the button.

`onClose` method sets `showDialog` as false and on the other hand is passed as a callback to `RequestCertificateModal` described in subsection [4.5.2](#) to be used to close the dialog from inside of the modal component.

4.5.2 RequestCertificateModal

This component renders a dialog containing the form used to request or import a new certificate. It uses Patternfly's [Modal](#) component for the dialog itself.

A class `ct-form` imported and saved in `lib/form-layout.scss` is used for whole form. It specifies design of the form and layout of the form for row labels and row content.

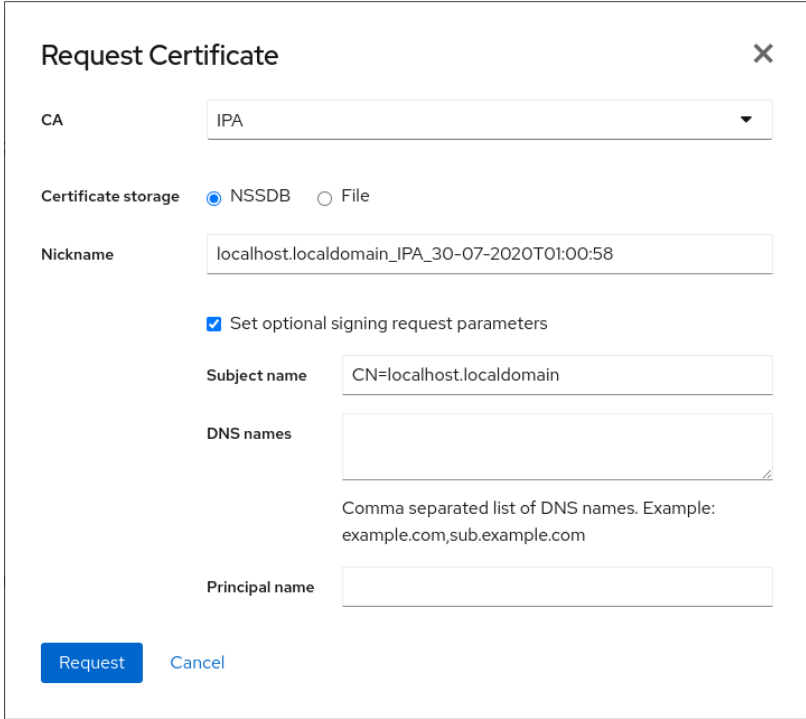


Figure 4.4: A screenshot of a `RequestCertificateModal` in „request“ mode

Its state has these properties:

- `_mode` - determines which functionality the dialog server. This property determines if some parts of form are not shown. Acceptable values for this property are strings „import“ or „request“.

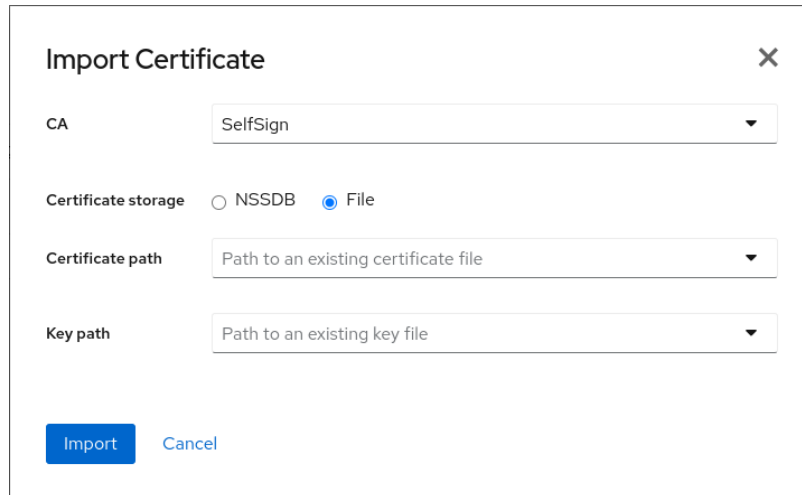


Figure 4.5: A screenshot of a RequestCertificateModal in „import“ mode

- **_hostname** - the name of host server in form of string. This is used to pre-generate some form values, such as nickname in component `NicknameRow` described in section 4.5.6
- **_userChangedNickname** - a boolean flag used to know if user changed the pre-generated value of nickname.
- **ca** - a string value of a certificate authority. The default value is the nickname of first certificate authority in a list of CAs provided by `certmonger`. It is set by `CAsRow` described in section 4.5.4
- **storage** - a string value determining the storage type of a new certificate. Acceptable values are „nssdb“ for NSS database and „file“ for storing certificate and key-pair file somewhere on file system. If user is requesting a new certificate, default value is „nssdb“, if user is importing an existing certificate, default value is „file“. It is set by `StorageRow` described in section 4.5.5
- **nickname** - a string value used as a nickname for NSS database. In case of file storage type, this property is irrelevant. It's value is pre-generated as a combination of host name, default certificate authority and date. It is set by `NicknameRow` described in section 4.5.6
- **certFile** - a string determining a file path where to store newly generated certificate file. In case of NSS database storage type, this property is irrelevant. It is set by `CertFileRow` described in section 4.5.7
- **keyFile** - a string determining a file path to existing key-pair to use when requesting a new certificate or where to store newly generated key-pair file. In case of NSS database storage type, this property is irrelevant. It is set by `KeyFileRow` described in section 4.5.8
- **signingParameters** - a boolean value determining if user wants to specify additional values for a signing request. It is set by `SetSigningParametersRow` described in section 4.5.9

- **subjectName** - an arbitrary and optional string value to used for common name (aka CN) of certificate. It's pre-generated as a „CN=hostname“ using server's hostname. It is set by **SubjectNameRow** described in section [4.5.10](#)
- **dnsName** - an optional string value to used for a list of DNS names of certificate. It can be only one name, or multiple names separated by a comma. This formatting is enforced in **onValueChanged** method. It is set by **DNSNameRow** described in section [4.5.11](#)
- **principalName** - an optional string value used as a princinal name for Kerberos. It is set by **PrincipalNameRow** described in section [4.5.12](#)

It consists of several methods:

- **componentDidMount** - a standard React method called when component first mounts. In this case, it's used to fetch the hostname of the server and save it into the state store to **_hostname** property.
- **onValueChanged** - a helper method used to do additional logic when property of some state changes. It is used to update the nickname and subject name when server hostname is fetched, update state **_userChangedNickname** property when nickname changes, update nickname when certificate authority is selected or to check if **dnsName** state property is a string of multiple dns names divided by a comma.
- **onAddError** - a simple wrapper to add error's name and message into state.
- **onRequest** - this method is called when user clicks on „Request“ button. It prepares parameters filled by user and submits them to the **addRequest**

The `<hr />` HTML tag is used to divide form into 3 groups of related rows. First group deals with choosing CA authority, second group deals with choosing a storage and third group deals with setting additional signing request parameters. It's form has following composition. It has been simplified in order to not show unnecessary implementation details:

```
<form className="ct-form">
  <CARow />
  <hr />
  <StorageRow />
  if state.storage == "nssdb"
    <NicknameRow />

  if state.storage == "file"
    <CertFileRow />
    <hr />
    <KeyFileRow />

  <hr />
  if state._mode == "request"
    <SetSigningParametersRow />
    if state.signingParameters
      <SubjectNameRow />
```

```
        <DNSNameRow />
        <PrincipalNameRow />
</form>
```

4.5.3 ResubmitCertificateModal

The implementation of this dialog is similar to that of `RequestCertificateModal` described in 4.5.2. It also renders a dialog Patternfly's `Modal` using containing with a form for specifying certificate properties.

Class `ct-form` is imported and saved in `lib/form-layout.scss` and used for whole form.

Its state has these properties:

- `ca` - a string value of a certificate authority. This value is filled with a certificate authority name which issued the resubmitted certificate. It is set by `CAsRow` described in section 4.5.4
- `storage` - a string value determining the storage type of the resubmitted certificate. This value is filled by the storage of the resubmitted certificate and cannot be changed. It is only used when resubmitting a certificate. It is set by `StorageRow` described in section 4.5.5
- `nickname` - a string value used as a nickname for NSS database. This value is filled by the nickname of the resubmitted certificate and cannot be changed. It is only used when resubmitting a certificate with NSSDB storage. It is set by `NicknameRow` described in section 4.5.6
- `certFile` - a string with a file path to a location of a resubmitted certificate. A certificate at this location will be replaced by a new certificate. It cannot be changed. It is only used when resubmitting a certificate with FILE storage type. It is set by `CertFileRow` described in section 4.5.7
- `keyFile` - a string with a file path to a location of a key-pair used for a resubmitted certificate. The key-pair at this location will used again when generating a new signing request. It cannot be changed. It is only used when resubmitting a certificate with FILE storage type. It is set by `KeyFileRow` described in section 4.5.8
- `subjectName` - a string value to used for common name (aka CN) of certificate. Prefilled by common name of the resubmitted certificate. It is set by `SubjectNameRow` described in section 4.5.10
- `dnsName` - a string for a list of DNS names of certificate. Prefilled by hostnames of the resubmitted certificate and formatted into a a string where each dns name is separated by a comma. This formatting is enforced in `onValueChanged` method. It is set by `DNSNameRow` described in section 4.5.11
- `principalName` - a string used for a principal name for Kerberos. Prefilled by principal name of the resubmitted certificate. It is set by `PrincipalNameRow` described in section 4.5.12

It consists of several methods:

- `onValueChanged` - a helper method used to update the component's state. It's only additional logic is to enforce a formatting of `dnsName` state property as a string of multiple dns names divided by a comma.
- `onAddError` - a simple wrapper to add error's name and message into state.
- `onResubmit` - this method is called when user clicks the „Resubmit“ button. It prepares parameters filled by user and submits them to the `modyfiRequest`, which modifies the current certificate. After this operation is done successfully it calls `/verb|resubmitRequest|`, which resubmits the certificate request to the certificate authority.

Figure 4.6: A screenshot of a `ResubmitCertificateModal` used to resubmit a certificate

Compared to the `RequestCertificateModal` described in 4.5.2 it doesn't use `NicknameRow`, `CertFileRow` or `KeyFileRow` which allow user to input values because in this case nickname, certificate file path and key-pair file path these are only read-only properties. It's form has following composition. It has been simplified in order to not show unnecessary implementation details:

```
<form className="ct-form">
  if storage === "nssdb"
    <label>Database path</label>
    <samp>{certificateDatabasePath}</samp>
    <label>Nickname</label>
    <samp>{certificateNickanme}</samp>

  if storage === "file"
    <label>Certificate file</label>
    <samp>{certificateFilePath}</samp>
```

```

        <label>Key file</label>
        <samp>{keyFilePath}</samp>

    <hr />
    <CARow />
    <hr />
    <SubjectNameRow />
    <DNSNameRow />
    <PrincipalNameRow />
</form>

```

Each row of the form has its own stateless component. Each row consists of label and way of gathering input from the user.

4.5.4 CARow

This row is used to select a certificate authority which the certificate request will be submitted to. It uses cockpit's own `Select` component, which is a wrapper for traditional `<select>` and `<option>` HTML tags. A list of certificate authorities provided by certmonger is mapped as options for this select. This row sets `ca` state property.

4.5.5 StorageRow

This is a pair of radio buttons where user can choose what type of storage they want to use for their certificate. Available options are „NSSDB“ and „File“. Based on which option user chooses, next few rows are rerendered. If they choose NSSDB the `NicknameRow` is rendered. If they choose File option the `CertFileRow` and `KeyFileRow` are rendered.

4.5.6 NicknameRow

This row is rendered only if NSS database is chosen for storage as it sets a nickname to be used to identify a certificate in NSSDB. It uses Patternfly's `TextInput`. This row sets `nickname` state property.

4.5.7 CertFileRow

Here user sets a file path where certificate file will be stored once provided by certificate authority. It uses cockpit's own `FileAutoComplete` component, which is aware of filesystem on a server, not a client, which is convenient for this situation. If no path is specified, a red alert row will be shown beneath. This row sets `certFile` state property.

4.5.8 KeyFileRow

Here user sets a path to the key-pair file. If user inputs path to an existing file, certmonger will use this file as a key-pair for a new certificate. If user inputs a path to a non-existing file, certmonger will generate the key-pair file and save it at the given location. If user is importing a certificate, they must provide a path to an existing file. It also uses cockpit's own `FileAutoComplete` component. Same as before, if no path is specified, a red alert row will be shown beneath. This row sets `keyFile` state property.

4.5.9 SetSigningParametersRow

A simple checkbox where user can choose if they want to set signing request parameters. If so, rows specifying subject name, dns name and principal name are shown underneath and indented to the right to represent the hierarchy of those rows related to this checkbox. If user decides not to specify signing parameters, certmonger will automatically generate some of those parameters. This row sets `setSigningParameters` state property.

4.5.10 SubjectNameRow

A text input field where user can specify subject name such as common name. It uses Patternfly's `TextInput`. This row sets `subjectName` state property.

4.5.11 DNSNameRow

A text area input field where user can specify DNS names for subject alternative names⁶. It uses Patternfly's vertically re-sizable `TextArea`. This row sets `DNSName` state property.

4.5.12 PrincipalNameRow

A text input field where user can specify principal name to be used for Kerberos. It uses Patternfly's `TextInput`. This row sets `principalName` state property.

4.6 Removing a certificate

This implements the functionality of removing a certificate. It's important to remember that there are differences between removing a certificate stored in NSS database and a certificate stored in a file on file system. This component renders a dialog. Depending on which storage type is chosen, the dialog has different body. Let's explain certificate removal for each storage type.

NSSDB: In this case, only read-only information which identifies a certificate for the user. There is a row which shows a path for NSSDB and a row which shows a nickname used to identify a certificate in NSSDB. This way user know which certificate in which database is being removed. **File storage:** Here are also two rows of read-only information which identifies a certificate for the user, but there is also a row which allows user to choose additional action of removing files associated with a certificate. There is a row which shows a location of certificate file, a row which shows a location of key file and a row with a checkbox which allows user to choose if they also want to delete certificate and key files.

It uses Patternfly's `Modal` component to render the dialog and `Button` component.

Its state has only one property:

- `deleteFiles` - a boolean flag which says if user also wants to delete associated certificate files. It's by default set to false and is only relevant if certificate which is being removed is stored in files and not in NSS database.

The class of this component has several methods:

- `onValueChanged` - a helper method used to update the component's state. It's mainly used as a callback for changing a state from `<input>` checkbox.

⁶An extension to X.509 certificates allowing user to secure additional domain names with same certificate

- `onRemove` - a callback which is called when user confirms deletion of a certificate. This is where most of the logic for removing a certificate is taking place. If user decided to also delete associated certificate files, `cockpit.file` API is called to delete the key file, then delete the certificate file and then function `removeRequest` is called to remove the certificate from `certmonger`. All of these actions are asynchronous, but are chained using JavaScript's `Promise.prototype.then()`⁷, meaning that next action is done only after previous one is successfully completed. If the associated certificate are not to be deleted, the certificate is only removed from `certmonger`. After all this, `appOnValueChanged` is called to manually update the list of certificates in application's state, because `certmonger` doesn't emit any signal when certificate is removed.

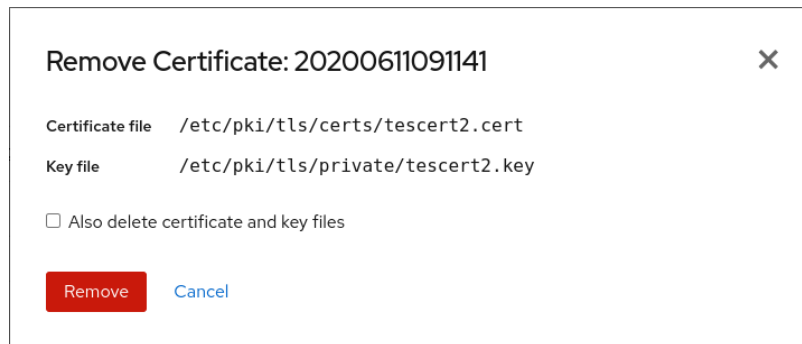


Figure 4.7: A screenshot of a `RemoveModal` used to remove a certificate

4.7 Error handling

Most of the error handling is done in `src/app.jsx`. A list of alerts is kept in state of `Application` component. Each alert is an object consisting of 2 properties: error title and error message. Helper methods are defined to manage these alerts:

- `addAlert`: this method takes an error title and an error message as arguments and pushes them into the list of alerts.
- `removeAlert`: this method takes index as an argument and removes item from the list of alerts with same index.

The main idea is that method `addAlert` is passed to each component as a callback which triggers an action which could possibly cause an exception.

Patternfly's [AlertGroup](#) and [Alert](#) components for rendering errors.

⁷A method which takes arguments in form of callbacks for success or failure, allowing programmer to specify what should be done after success or failure.

Chapter 5

Testing

This application is based on Cockpit's starter-kit¹. It provides an easy way to write tests in Python by just adding test functions into `/test/check-application`. „Run make check to build an RPM, install it into a standard Cockpit test VM (centos-7 by default), and run the test/check-application integration test on it. This uses Cockpit's Chrome DevTools Protocol based browser tests, through a Python API abstraction.“^[13] These tests are also part of testing for continuous integration² for upstream project. Therefore these tests are run every time a contributor creates a Github pull request³ for this project. Continuous integration infrastructure is already provided by Cockpit.

It provides two objects:

- `self.machine` - It's used as interface to a virtual machine which tests are being run on. It offer function `self.machine.execute` to spawn commands on the virtual machine's sytem.
- `self.browser` - It's used as interface to a browser which runs the application inside the virtual machine. It offer functions such as `self.machine.wait_present` to check presence of element in DOM, `self.machine.click` to click on an element in DOM, `self.machine.wait_in_text` to check presence of text in an DOM's element and many more functions.

Below is a summary of all unit tests which were written for this application.

5.1 Empty state

Implemented in function `testEmptyState`. Tests a functionality of implementation described in section 4.2. It can be summarized in these steps:

1. It stops `certmonger.service` by spawning command:

```
systemctl stop certmonger.service
```

¹<https://github.com/cockpit-project/starter-kit/>

²A practice of merging code of other developers continuously

³Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.^[9]

and loads the application page to check if page tries to start the service automatically.

2. It stops the service again to check if this time the application doesn't try to start the service again and instead it renders `EmptyState` component.
3. Tries to start the service from UI and checks if UI reacts to change of service's state.
4. Stops and starts the service to see if a starting of the service from command line is reflected in the UI.

5.2 No certificates available

Implemented in function `testNoCertificates`. Test checks a correct behaviour of the application if no certificate is available on a system. If no certificate was created before the test was started, a text which is telling a user that no certificates are available should be rendered.

5.3 Warning of expired certificates

Implemented in function `testWarningExpiredCert`. Tests a correct warning of expired certificate or a certificate which is going to expire. It's implementation described in section [4.3](#).

The test's structure can be described in following steps:

1. It creates a basic self-signed certificate with a command:

```
selfsign-getcert request
```

and it checks that no expiration warning is shown.

2. It changes certmonger's configuration to issue self-signed certificates only for 2 weeks, requests a self-signed certificate and checks that a warning about a certificate going to expire soon is present.
3. It changes certmonger's configuration to issue self-signed certificates only for 1 second, requests a self-signed certificate and sleeps for 4 seconds. Then it checks that an error about an expired certificate is present.

5.4 Modifying auto-renewal

Implemented in function `testModifyAutorenewal`. Tests a correct functionality of turning off and turning on a certificate's automatic renewal. It's implementation described in section [4.3.1](#).

The test's structure can be described in following steps:

1. It creates a basic self-signed certificate with a command:

```
selfsign-getcert request
```

2. It reads a value of a checkbox which sets automatic renewal and then negates this value.

3. It checks if value of a checkbox was negated and co-checks this change by looking at certificate's „auto-renew“ property output of a command:

```
selfsign-getcert list
```

5.5 NSSDB stored certificate

Implemented in function `testNssdbCert`. Tests a correct representation of NSSDB stored certificate. It checks if it correctly shows expected properties of a certificate as described in section 4.3.

The test's structure can be described in following steps:

1. It creates a basic self-signed certificate with a command:

```
selfsign-getcert request
```

2. It compares rendered values with expected values for tabs „General“, „Key“ and „Cert“.

5.6 File stored certificate

Implemented in function `testFileCert`. Does the same thing as `testNssdbCert` described in section 5.5 but with values expected for a file stored certificate.

5.7 Updating after receiving D-Bus signal

Implemented in function `testDbusPropertyChanged`. Tests correct subscription to a D-BUS signal `PropertyChanged`. Everytime any property of a certificate changes by some external actions, D-BUS should emit a signal and the UI should update accordingly.

The test's structure:

1. Create a basic self-signed certificate with a command:

```
selfsign-getcert request
```

2. Save certificate's validity value shown in the UI
3. Renew a certificate by running a command:

```
selfsign-getcert resubmit
```

This should result in certificate's updated validity.

4. Check a certificate's validity has changed.

5.8 Removing a certificate

Implemented in function `testRemoveCert`. Tests removing a certificate. Implementation of this functionality is described in section 4.5.3

The test's structure:

1. Create a basic NSSDB stored self-signed certificate.
2. Remove it using functionality exposed by the UI. Check an option to remove associated certificate files is not available, since it's NSSDB stored certificate.
3. Check the certificate is no longer present in the UI.
4. Create a basic file stored self-signed certificate.
5. Remove it using functionality exposed by the UI. Check an option to remove associated certificate files is available, since it's file stored certificate, but do not choose it.
6. Check the certificate is no longer present in the UI but files are still available on system.
7. Create a basic file stored self-signed certificate.
8. Remove it using functionality exposed by the UI. Also choose option to delete associated files.
9. Check the certificate is no longer present in the UI and files were deleted.

5.9 Importing a certificate

Implemented in function `testImportCert`. Tests importing a certificate. Implementation of this functionality is described in sections [4.5.1](#) and [4.5.2](#).

As importing a certificate is action which needs to be tested multiple times with different parameters. Therefore a class `ImportCertDialog` for this repetitive testing was created. This class has following methods:

- **execute**: Executes a single test instance for importing a certificate. It calls following methods in this order: `open`, `fill`, `create`, `verify_frontend`, `verify_backend`, `cleanup`.
- **open**: Opens a dialog for importing a certificate
- **fill**: Fills the dialog with inputs
- **create**: Clicks on „Import“ button which imports a certificate
- **verify_frontend**: Checks if imported certificate is present in the UI with correct properties.
- **verify_backend**: Checks if imported certificate has equal properties as those shown by CLI tool `selfsign-getcert`.
- **cleanup**: Removes an imported certificate

In order to have a certificate that can be imported, commands are spawned which create and then remove a certificate. They do not however cleanup certificate files, which can be then used for importing a certificate.

5.10 Requesting a certificate

Implemented in function `testImportCert`. Tests requesting a new certificate. Implementation of this functionality is described in sections [4.5.1](#) and [4.5.2](#).

Same as with importing a certificate, this is an action that needs to be tested multiple times with different parameters. Therefore a class `RequestCertDialog` was created. Although this class has the same methods as `ImportCertDialog` from the previous section, their implementation differs significantly. It has following methods:

- `execute`: Executes a single test instance for creating a certificate. It calls following methods in this order: `open`, `fill`, `create`, `verify_frontend`, `verify_backend`, `cleanup`.
- `open`: Opens a dialog for requesting a certificate
- `fill`: Fills the dialog with inputs
- `create`: Clicks on „Create“ button which requests a certificate
- `verify_frontend`: Checks if new certificate is present in the UI with correct properties.
- `verify_backend`: Checks if new certificate has equal properties as those shown by CLI tool `selfsign-getcert`.
- `cleanup`: Removes a certificate

This test runs 4 different test instances, each with a different aim. First, it tries to run a test without providing any inputs, to see if values available in a dialog for requesting a certificate by default will result in a proper certificate. Then it tries to request an NSSDB-stored certificate, file stored certificate and in the end, it tries to request a certificate with additional signing properties: subject name, DNS name, and principal name.

Chapter 6

Conclusion

This work provides a tool with a user interface to manage server's certificates. During the designing of the application, it was always taken into account to present functionality to a user who may be a layman in a topic of X.509 certificates, an administrator who is mainly working with Windows, or a beginner administrator. These problems were solved in various ways. The application tries to navigate the user to a desirable outcome that would satisfy most of the users. This was achieved by setting default options for inputs every time it was possible. Another way was trying to not overcrowd any component or modal with too many inputs or data.

It's aim was also to provide a solution that would fit into Red Hat's infrastructure. This was achieved, as the plugin was presented to accepted by Cockpit's team into its project portfolio and in the future, its repository will be moved under cockpit-project. A process of releasing the project as a package into Fedora is ongoing and in near future, there will be a `cockpit-certificates` package available.

Project is currently available at its Github repository¹. There are plans to extend it. Currently, it provides solutions mainly for FreeIPA infrastructure, as that's Red Hat's default solution for identity management, but it might be enlarged by another authority such as Let's Encrypt².

¹<https://github.com/skobyda/cockpit-certificates>

²Popular open-source and free certificate authority

Bibliography

- [1] *Overview - certmonger* [online]. [cit. 2020-06-28]. Available at: <https://pagure.io/certmonger>.
- [2] *Cockpit* [online]. [cit. 2020-06-26]. Available at: <https://cockpit-project.org/>.
- [3] *What is CSR* [online]. [cit. 2020-06-24]. Available at: <https://www.appviewx.com/education-center/certificate-request/what-is-csr/>.
- [4] *Dbus* [online]. [cit. 2020-06-28]. Available at: <https://www.freedesktop.org/wiki/Software/dbus/>.
- [5] *D-Bus Tutorial* [online].
- [6] *DBus Overview* [online]. [cit. 2020-06-29]. Available at: https://pythonhosted.org/txdbus/dbus_overview.html.
- [7] *About JavaScript* [online]. [cit. 2020-06-27]. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.
- [8] *About PatternFly* [online]. [cit. 2020-06-28]. Available at: <https://www.patternfly.org/v4/get-started/about>.
- [9] *About pull requests* [online]. [cit. 2020-07-28]. Available at: <https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests>.
- [10] *React - A JavaScript library for building user interfaces* [online]. [cit. 2020-06-28]. Available at: <https://reactjs.org/>.
- [11] *Planning How to Deploy Red Hat Certificate System* [online]. [cit. 2020-06-20]. Available at: https://access.redhat.com/documentation/en-us/red_hat_certificate_system/9/html/planning_installation_and_deployment_guide/planning_how_to_deploy_rhcs.
- [12] *Renewing Certificates* [online]. [cit. 2020-07-20]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/linux_domain_identity_authentication_and_policy_guide/cert-renewal.
- [13] *Automated Testing* [online]. [cit. 2020-07-28]. Available at: <https://github.com/cockpit-project/starter-kit/blob/master/README.md#automated-testing>.
- [14] XENITELLIS, S. S. *The Open-source PKI Book: A guide to PKIs and Open-source Implementations*. 1999.