MUNI

Projekt VI20202022164

# Pokročilá orchestrace bezpečnosti a inteligentní řízení hrozeb („ORION“)

## Kolaborativní prostředí pro členy bezpečnostních týmů (R3)

Typ výsledku: R – Software

Počet listů: 106 + krycí list

**Masarykova univerzita**

Žerotínovo nám. 617/9, 601 77 Brno, Česká republika
T: +420 549 49 1111, E: info@muni.cz, W: www.muni.cz

# Obsah

# 1. Technická dokumentace

## 1.1. Vývoj uživatelského rozhraní

Tvorba grafického uživatelského rozhraní (GUI) je časově náročným a nákladným úkolem. V případě komplexních systémů a nástrojů však představuje GUI pouze pomyslnou špičku ledovce. V kontextu bezpečnosti ICT infrastruktur je i při vynaložení nepřiměřených nákladů takřka nadlidským úkolem vytvořit univerzální uživatelské rozhraní, které by pokrylo všechny informační, reakční, či analytické scénáře a situace, se kterými se setkávají členové bezpečnostních týmů, resp. další relevantní uživatelské role. Vybrané (bezpečnostní) nástroje mají navíc svá uživatelská rozhraní, která jsou potenciálně ověřená zpětnou vazbou uživatelů a jejich požadavků za roky používání.

Na druhou stranu je v komplexní doméně bezpečnosti ICT infrastruktur vhodné mít při práci v týmu výchozí centrální bod pro rychlou orientaci jeho jednotlivých členů v aktuální situaci a pro podporu jejich vzájemné spolupráce při řešení nastalých situací. Cílem výsledku bylo v souladu s Paretovým pravidlem identifikovat a podpořit 20 % těch činností a úkonů, které pokryjí až 80 % často se opakujících scénářů a situací. V průběhu řešení bylo proto uživatelské rozhraní průběžně upravováno jak na základě požadavků uživatelů, tak v souladu se zvolenými případy užití v podobě orchestračních scénářů [1].

### Uživatelské požadavky dle uživatelských rolí

V rané fázi projektu proběhly iniciální řízené interview a rešerše literatury s cílem získat alespoň základní představu o požadavcích uživatelů. Bezpečnostní operativa se dotýká různých uživatelských rolí, a jednotliví uživatelé spolu nezřídka spolupracují. Proto je potřebné, aby každý z nich měl přehled o aktuálním stavu spravované ICT infrastruktury a služeb, resp. aby měl přístup ke klíčovým informacím při řešení nastalých situací (případů). Jednotlivé uživatelské požadavky se pohybovaly v rozpětí jak velmi konkrétních, tak velmi obecných.

Výchozí plán řešitelského týmu zaměřit se při implementaci uživatelského rozhraní nejdříve na přehledové obrazovky (dashboardy) byl výrazněji korigován poté, co se ukázalo, že pro členy bezpečnostního týmu je mnohem zásadnější agenda pro řízení případů, a proto pro ni bude nutné vytvořit množství podpůrné funkcionality. Zároveň bylo nutné do uživatelského rozhraní postupně přidávat nové prvky na základě postupného vývoje orchestračních scénářů. V průběhu řešení projektu ovšem také vznikly prototypy pohledů, které nakonec nejsou součástí konečného výsledku po vyhodnocení jejich nižší priority. Níže přinášíme výčet uživatelských požadavků, na které se řešitelský tým rozhodl ve výsledku reagovat v souladu s pravidlem 80/20 diskutovaným výše.

#### L4 vedoucí CSIRT týmu

- Zobrazení počtu a trendu případů dle jejich stavu za daný časový interval – informace o počtu případů za předem definovaný interval.
- Zajištění základního přehledu o aktivech – informace o vybraných typech aktiv.
- Shrnující pohledy na data v externích systémech.

#### L1-L3 specialisté

- Zobrazení počtu a trendu případů dle jejich stavu za daný časový interval – informace o počtu případů za předem definovaný interval.
- Zobrazení počtu přiřazených případů – analogicky předchozímu pro případy, které má daný pracovník přiřazeny k řešení.
- Zobrazení detailních informací o minulých i aktuálních případech – komplexní pohled na případ a relevantní data, související postup řešení a přijatá protiopatření.
- Manipulace s případem – přidávání, odebíraní a aktualizace informací a vazeb.
- Pohled na pozici případu v kontextu procesu pro řízení životního cyklu hrozeb.
- Detailní pohled pro prohledávání datového skladu.
- Zajištění základního přehledu o aktivech – informace o vybraných typech aktiv.

- Shrnující pohledy na data v externích systémech.

### Správce ICT

- Zobrazení přehledových informací o minulých i aktuálních případech ve spravované části infrastruktury.
- Zajištění základního přehledu o aktivech ve spravované části infrastruktury – informace o vybraných typech aktiv.

### Operátor uživatelské podpory

- Přehledná L0 dokumentace.
- Intuitivní formulář pro hlášení potenciálního bezpečnostního incidentu.

### Uživatel ICT

- Intuitivní formulář pro hlášení potenciálního bezpečnostního incidentu.

### Obecné požadavky

- Zobrazovaná data budou dynamicky aktualizována při změně stavu (např. v případě přehledu síťového provozu to může být v řádu jednotek minut).
- Výchozí rozlišení obrazovek bude min. 1920x1080 pixelů, předpokládá se také zobrazování na monitorech stolních počítačů a laptopů.
- Na základě vytvořeného micro-frameworku bude možné s přijatelným úsilím implementovat nové typy pohledů za využití existujících komponent.

## Kategorie pohledů v kolaborativním prostředí

Prostřednictvím analýzy požadavků a diskuse s vybranými reprezentanty uživatelských rolí jsme identifikovali hlavní vzorce práce s informacemi, z nichž následně vzešly čtyři kategorie pohledů. Většina požadavků vychází z potřeby získání rychlého přehledu o situaci, či naopak potřeby detailní analýzy daného konceptu (např. aktivum, událost, případ, hrozba, blokace). V případě L1-L3 specialistů vyvstává potřeba základní interakce s daty pro získání detailních informací (tzv. drill-down) a dále také sdílení a úprava informací v rámci jednoho případu. Z analýzy uživatelských požadavků dále vyplývá, že většina případů užití zahrnuje přístup k datům v režimu čtení, případně základní interakce s daty (např. jednoduché filtry, řazení). Výsledkem jsou čtyři kategorie pohledů, jejichž konkrétní instance byly postupně realizovány ve vyvíjeném kolaborativním prostředí.

- *Obecný pohled* – zobrazuje souhrnné a přehledové informace o různých konceptech a poskytuje tak všeobecný přehled o jejich množství a základním stavu.
- *Detailní pohled* – zobrazuje podrobné informace o stavu, historii a konkrétních parametrech daného konceptu, případně zobrazuje základní vazby na ostatní koncepty.
- *Pokročilý pohled* – specializované zobrazení dat souvisejících s konkrétním konceptem. Může zobrazovat pokročilé vazby mezi koncepty. Podpora základní editace.
- *Analytický pohled* – podpora pokročilé manipulace s daty souvisejících s konkrétním konceptem. Dovoluje vytvářet a aktualizovat vazby mezi koncepty.

## 1.2. Řízení případů

Výsledek vychází z principů používaných při řízení případů (Case Management) [2]. Řízení případů je způsob organizace práce založený na koncentraci všech relevantních informací k nějaké situaci do jednoho místa, tj. případu (pracovní složky). Kolaborativní prostředí se inspiruje v jiných oblastech, kde je organizace práce formou vedení spisu ustáleným konceptem (lékařský spis, policejní případ). Výsledek měl za cíl tento koncept převést do prostředí kyberbezpečnosti a adresovat jeho specifika.

Případ je pro řešení dané situace hlavním zájmovým bodem. Jednotlivé případy tak agregují všechna relevantní data a vykonané činnosti týkající se dané situace, umožňují základní drill-down a poskytují podporu pro základní organizační, technické a procesní postupy. Zároveň však poskytují prostor pro řešení neznámých a nestandardních situací za podpory historické evidence všech (manuálních i automatických) akcí vykonaných v rámci řešení případu. Níže přinášíme stručný souhrn používaných konceptů.

### Základní koncepty

#### Případ (Case)

Případ je hlavní datovou entitou kolaborativního prostředí, sledující vznik, průběh a řešení určité bezpečnostní situace (např. incidentu) a sdružující veškeré informace relevantní k dané situaci. Případ se otevírá manuálně nebo automatizovaně na základě bezpečnostní události. Je řešen inkrementálně, dle pracovního postupu, který je mu přiřazen na základě jeho charakteru. Uzavírá se vyřešením všech úkolů pracovního postupu.

#### Pracovní postup a šablona pracovního postupu (Workflow a Workflow template)

Pracovní postup popisuje průběh řešení případu a určuje jeho aktuální stav (fáze k řešení, právě řešené fáze a vyřešené fáze). Postup vzniká manuálním vytvořením, nebo na základě šablony pracovního postupu. Pracovní postup se dále dělí do fází, které obsahují úkoly. Pracovní postup je vyřešen v okamžiku vyřešení všech jeho fází.

Šablona je vzor, popisující možný průběh řešení případu dle jeho charakteristiky. Definuje jednotlivé fáze, jejich úkoly, typy úkolů a návaznosti úkolů. Pracovní postup sám o sobě nemá provázané entity, nicméně jeho detailní pohled obsahuje agregaci entit provázaných s úkoly náležícími pracovnímu postupu (přesněji fázím náležícím pracovnímu postupu).

#### Fáze pracovního postupu (Stage)

Fáze je milníkem pracovního postupu a oddělovačem jeho logických celků. Fáze obsahuje úkoly a je vyřešena v okamžiku vyřešení všech jejich úkolů. V rámci jedné fáze jsou úkoly prováděny v sérii. Podobně jako pracovní postup, ani fáze neobsahuje provázané entity, pouze jejich agregaci na základě přiřazených úkolů.

#### Úkol (Task)

Úkol je atomickou jednotkou práce vykonávané při řešení daného případu. Řešitelem případu může být člen bezpečnostního týmu, nebo stroj, který jej řeší automatizovaně. Každý úkol očekává vstup předem definovaného typu, jenž je základem k řešení případu (např. pro úkol rozhodnutí o zablokování IP adresy by byla vstupem konkrétní IP adresa).

## 1.3. Shrnutí průběhu řešení

### Etapa I

V rámci Etapy I byla navržena celková vysokoúrovňová architektura a funkcionalita nástroje pro kolaborativní prostředí založená na poznatcích z oblasti přehledových obrazovek (dashboards), které dovolí jednotlivým uživatelským rolím rychlou orientaci v aktuální situaci, a dále pak z oblasti řízení případů (case management). Řízení případů je způsob organizace práce založený na koncentraci všech relevantních informací k řešení dané situace (např. indikované hrozby, či incidentu) do jednoho místa, tj. případu (pracovní složky), který se tak stává pro danou situaci hlavním zájmovým bodem.

Dále byl v rámci Etapy I proveden sběr uživatelských požadavků za účelem identifikace typických přehledových obrazovek a požadovaných pohledů dle jednotlivých uživatelských rolí. Na základě prvotní analýzy byly definovány čtyři kategorie požadovaných pohledů s různou úrovní složitosti uživatelského rozhraní a způsobů interakce se zobrazovanými daty. Jakožto základ pro budoucí uživatelské prostředí na straně front-end byl vytvořen jednoduchý mikro-framework na bázi podřazeného modulárního webového frameworku Angular. Za účelem ověření jeho vlastností a funkcionality byly vytvořeny základní příklady pohledů v kontextu ochrany ICT infrastruktury.

### Etapa II

Cílem Etapy II bylo vytvořit dodatečné pokročilé pohledy v kontextu ochrany infrastruktury, které by dovolovaly specializované zobrazení dat souvisejících s konkrétním konceptem. V rámci Etapy II byl tímto konceptem především „případ“ (case), jakožto kritický prvek pro zamýšlený návrh a implementaci základní verze uživatelského prostředí pro podporu spolupráce a pro bezpečné sdílení kontextově obohacených dat o řešených incidentech a hrozbách.

Bylo tedy nutné navrhnout a implementovat jak specializovanou back-end komponentu (mikro-službu) pro spolupráci a sdílení informací a související datové mikro-služby, tak vlastní pohledy uživatelského rozhraní. V průběhu řešení výsledku se po prvotním ověření osvědčilo použití jazyka GraphQL pro definici vybraných HTTP API rozhraní, namísto použití tradičního REST rozhraní. Byl vytvořen prototyp komponenty pro spolupráci a sdílení informací v podobě databáze případů a souvisejících datových mikro-služeb pro prvotní napojení potřebných externích systémů a entit. Prototyp byl vyvinut v jazyce Python a jeho API je popsáno pomocí jazyka GraphQL. V jazyce Python byl také vyvinut specializovaný mikro-framework, který vývoj GraphQL aplikací výrazně ulehčuje (viz Výsledek č. 1 [3]). Dále byly vytvořeny nové komponenty uživatelského rozhraní, jež byly sdruženy do vhodných pokročilých pohledů. Vývoj uživatelského rozhraní výrazně urychlilo právě použití GraphQL API rozhraní a možnost automatického generování částí kódu.

V souladu s plánem Etapy II vzniky pokročilé pohledy v kontextu ochrany infrastruktury, které dovolují specializované zobrazení dat souvisejících s konceptem případu. Klíčovým je v tomto ohledu pokročilý pohled na detail bezpečnostního případu.

### Etapa III

Cílem Etapy III byla finalizace implementace uživatelského prostředí pro podporu spolupráce a pro bezpečné sdílení kontextově obohacených dat o řešených incidentech a hrozbách. Relativně přímočarým plánem bylo především finalizovat všechny potřebné agendy „Uživatelského rozhraní“ na straně front-end i back-end, rozšířit funkcionalitu mikro-služby „Databáze případů“ a v neposlední řadě zaměřit pozornost na problematiku autentizace a autorizace uživatelů.

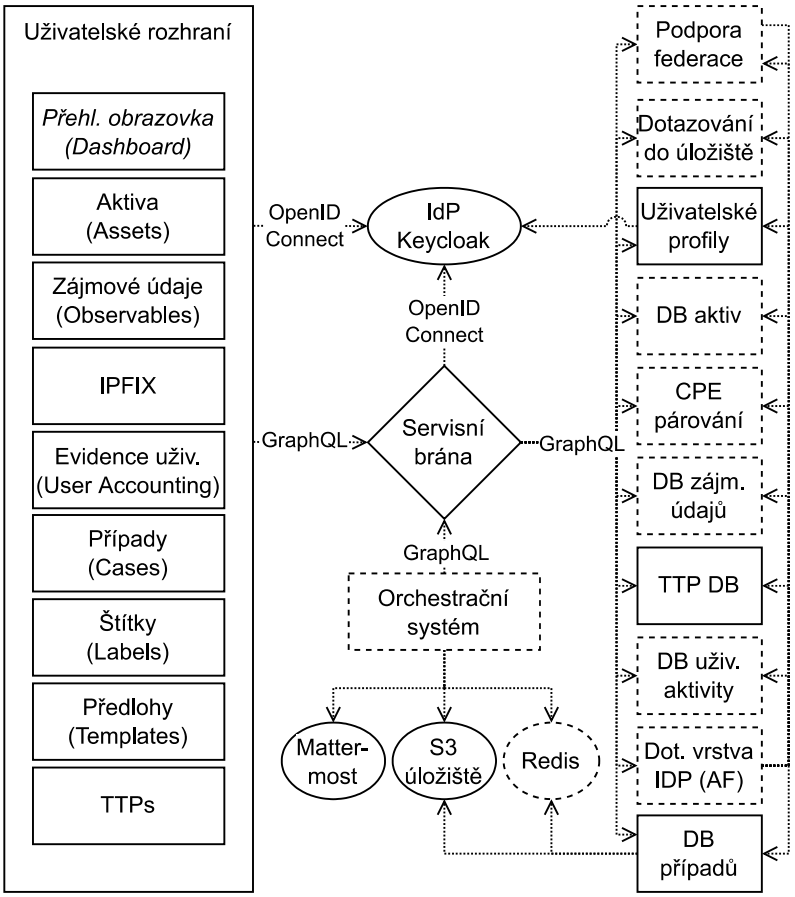
V souladu s výše uvedeným plánem došlo v rámci Etapy III k implementaci nové funkcionality pro všechny agendy a pohledy „Uživatelského rozhraní“, dále pak byla rozšířena mikro-služba „Databáze případů“ a mikro-služba „Databáze TTP“. Byla také vytvořena specializovaná autentizační a autorizační brána pro zabezpečení jednotlivých mikro-služeb na bázi standardu OpenID Connect a JWT (JSON Web Token). S tímto souvisí také vznik mikro-služby pro „Uživatelské profily“. U všech komponent byly

provedeny nutné úpravy v souvislosti s integrací s ostatními dvěma výsledky. „Uživatelské rozhraní“ realizuje spolu s „Databází případů“ (a přirozeně také ostatními výsledky) platformu pro sdílení kontextově obohacených informací o bezpečnostních incidentech a hrozbách.

V rámci Etapy III zároveň došlo k finalizaci zamýšleného výsledku a propojení všech jeho odpovídajících komponent z předchozích etap. Došlo také k vytvoření všech souvisejících softwarových artefaktů nezbytných k nasazení a provozu výsledku v orchestračním prostředí Kubernetes. Výsledkem č. 3 je tak *Kolaborativní prostředí pro členy bezpečnostních týmů (R3).* V rámci příslušných činností pro rok 2022 pak také došlo k provázání vytvořených výsledků do ucelené podoby.

## 1.4. Architektura výsledku

Obrázek 1 představuje celkovou architekturu výsledku, včetně všech vytvořených komponent, použitých rozhraní, protokolů, a datových formátů. Kolaborativní prostředí se skládá z *Uživatelského rozhraní a jeho agend* a souvisejících *komponent mikro-služeb*, především tedy *Databáze případů, Databáze TTP* a služby pro *Uživatelské profily*. Součástí výsledku je *Servisní brána*, která pro všechny mikro-služby projektu zajišťuje autentizaci a autorizaci uživatelů. Externí nástroj *Mattermost* realizuje podporu pro komunikaci uživatelů v reálném čase (více o nástroji lze nalézt v příručce k Výsledku č. 2 [1]).




*Obrázek 1. Celková architektura Výsledku č. 3 (R3)*

## 1.5. Komponenta Databáze případů

Komponenta databáze případů je zásadní pro spolupráci členů bezpečnostního týmu v kolaborativním prostředí. Koncept řízení případů je nosným principem pro sdílení dat a informací, kdy dochází ke koncentraci všech relevantních informací k nějaké situaci do jednoho místa, tj. případu (pracovní složky). Případ v rámci výsledku představuje centrální bod spolupráce nejen pro uživatele, ale také pro služby a komponenty ostatních výsledků.

### Případ (Case)

Případ je hlavní datovou entitou kolaborativního prostředí, sledující vznik, průběh a řešení určité bezpečnostní situace (např. incidentu) a sdružující veškeré informace relevantní k dané situaci. Případ se otevírá manuálně nebo automatizovaně na základě bezpečnostní události. Je řešen inkrementálně, dle pracovního postupu, který je mu přiřazen na základě jeho charakteru. Uzavírá se vyřešením všech úkolů pracovního postupu.

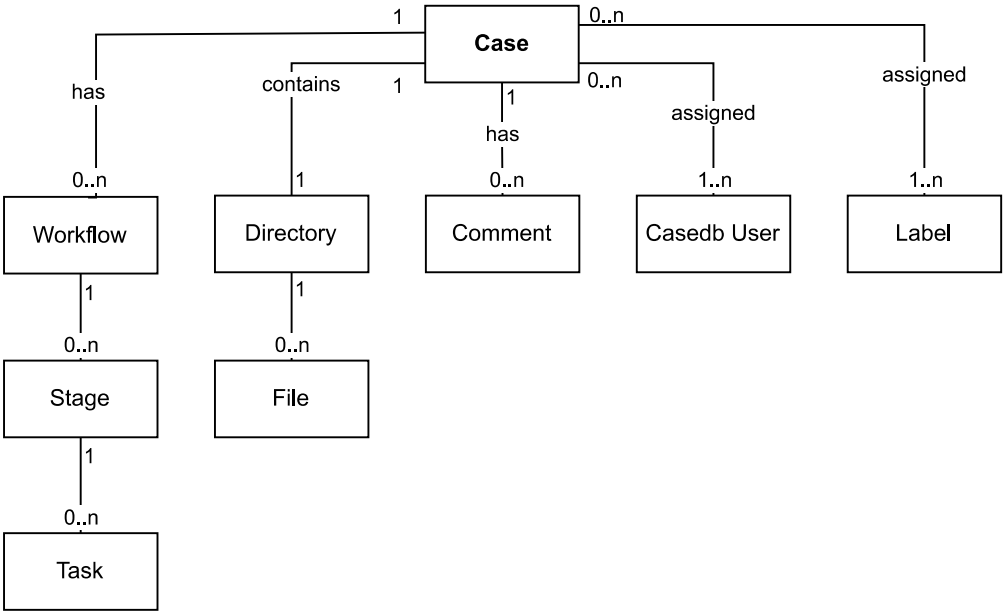
K dalším součástem případu patří jednoduchý souborový systém pro náhled a zabezpečenou práci s adresáři a soubory. Pro lepší zasazení případu do kontextu lze mezi případy vytvářet vztahy různého charakteru (podobnost, duplicita, návaznost, součást celku apod.). Řešitelé případu nebo ostatní uživatelé kolaborativního prostředí mohou případ komentovat. Případ sám o sobě nemá provázané entity, jako jsou aktiva, zájmové údaje, události, techniky, taktiky, procedury, uživatelé a externí tikety, nicméně obsahuje jejich agregaci na základě přiřazených úkolů.

Z hlediska návrhu grafického rozhraní se pohled dělí na hlavní obsah a obsah v postranním panelu. Postranní panel obsahuje přehled základních atributů případu (primitivní datové typy a pole), zejména pak těch, které je možné editovat. Hlavní obsah se dále dělí na několik záložek, kde ta první v pořadí je vždy přehledová a obsahuje komplexní grafické komponenty pro přímou práci na případu a jeho součástech. Následující záložky obsahují obecné pohledy na agregace provázaných entit s možností jednoduchého filtrování. Toto rozložení je konzistentní i v analytických pohledech na součásti případu, jako jsou pracovní postup, fáze a úkol.

Případ může nabývat následujících stavů.

- Otevřený (open),
- vyřešený (resolved),
- zamítnutý (rejected).

Nově založený případ nabývá stavu otevřený. Případ se stává vyřešeným, když jeden z jeho řešitelů provede akci vyřešit a zároveň jsou všechny jeho pracovní postupy, fáze pracovního postupu i úkoly vyřešené. Podobně se případ stává zamítnutým, když řešitel provede akci zamítnout.




*Obrázek 2. Doménový model entity Případ*

## Pracovní postup a šablona pracovního postupu (Workflow a Workflow template)

Pracovní postup popisuje průběh řešení případu a určuje jeho aktuální stav (fáze k řešení, právě řešené fáze a vyřešené fáze). Postup vzniká manuálním vytvořením, nebo na základě šablony pracovního postupu. Pracovní postup se dále dělí do fází, které obsahují úkoly. Pracovní postup je vyřešen v okamžiku vyřešení všech jeho fází.

Pracovní postup může nabývat následujících stavů.

- Probíhající (in progress),
- dokončený (done),
- zamítnutý (rejected).

Po vytvoření získává pracovní postup *probíhající* stav. V okamžiku, kdy jsou všechny fáze pracovního postupu v zamítnutém stavu, stává se i pracovní postup zamítnutým. Do stavu dokončený se pracovní postup přesouvá ve chvíli, kdy jsou všechny jeho fáze dokončené nebo zamítnuté (alespoň jedna fáze musí být dokončená).

Šablona je vzor, popisující možný průběh řešení případu dle jeho charakteristiky. Definuje jednotlivé fáze, jejich úkoly, typy úkolů a návaznosti úkolů. Pracovní postup sám o sobě nemá provázané entity, nicméně jeho detailní pohled obsahuje agregaci entit provázaných s úkoly náležícími pracovnímu postupu (přesněji fázím náležícím pracovnímu postupu).

Šablonu je možné vytvořit na základě existujícího pracovního postupu, exportovat a importovat ze souboru ve formátu JSON.

### Fáze pracovního postupu (Stage)

Fáze je milníkem pracovního postupu a oddělovačem jeho logických celků. Fáze obsahuje úkoly a je vyřešena v okamžiku vyřešení všech jejich úkolů. V rámci jedné fáze jsou úkoly prováděny v sérii. Podobně jako pracovní postup, ani fáze neobsahuje provázané entity, pouze jejich agregaci na základě přiřazených úkolů.

Fáze pracovního postupu může nabývat následujících stavů.

- Čekající (pending),

- probíhající (in progress),
- dokončený (done),
- zamítnutý (rejected).

Jelikož jsou fáze pracovního postupu prováděny v sérii, získává stav probíhající vždy pořadově první fáze. Všechny následující fáze získávají po vytvoření stav čekající. Do stavů dokončený, nebo zamítnutý se fáze přesouvají na základě stavu jejich úkolů. V případě, že jsou všechny úkoly fáze ve stavu zamítnuty, stává se i fáze samotná zamítnutou. Pokud je alespoň jeden úkol dokončený a všechny ostatní úkoly jsou dokončeny nebo zamítnuty, nabývá fáze stavu dokončena. V momentě, kdy se fáze stává dokončenou, přesouvá se následující fáze ze stavu čekající do stavu probíhající. V případě, že je fáze zamítnuta, jsou zamítnuty i všechny následující fáze.

## Úkol (Task)

Úkol je atomickou jednotkou práce vykonávané při řešení daného případu. Řešitelem případu může být člen bezpečnostního týmu, nebo stroj, který jej řeší automatizovaně. Úkol může obsahovat vstup určitého typu, jenž může být základem k řešení případu (např. pro úkol rozhodnutí o hromadném zablokování IP adres by byl vstupem seznam konkrétních IP adres). Výsledkem úkolu je výstup určitého typu (např. pro úkol hromadného zablokování IP adres by byl výstupem výsledek výběru konkrétních IP adres ze seznamu na vstupu).

Vstup i výstup se odlišují dle typu, který ovlivňuje grafické zobrazení komponenty pro zadání a validaci (např. vstup typu JSON může být validován dle přiloženého JSON Schema).

Každý úkol má jednoho či více následovníků. Vztahy mezi úkoly jsou ovlivněny příznaky jednotlivých úkolů. Následovníci úkolu bez příznaků mohou nabývat pouze vztahu „po dokončení“. Tento typ vztahu znamená, že po dokončení úkolu je otevřen jeho následovník. Úkoly s rozhodovacím příznakem (decision) mohou spustit různé úkoly na základě hodnoty rozhodnutí. Jejich následovníci nabývají vztahů „po dokončení“, „uzavřeno hodnotou pravda“, „uzavřeno hodnotou nepravda“. Vzniká tak průchod stromem úkolů. Posledním možným typem vztahuje je „blokuje“. Tohoto typu mohou nabývat následovníci úkolu s příznakem agregátor. Blokovaný úkol musí čekat na vyřešení všech úkolů, který jej blokují.

Úkol obsahuje přímé reference na provázané entity. Zatímco případ může být rozsáhlý a obsahovat tak velké množství agregovaných referencí, u úkolu je předpokládaný počet referencí spíše nižší. Tím se odlišuje i praktické použití referencí. V pokročilém pohledu případu je cílem referencí zejména získání přehledu a možnost průzkumu (pohyb po grafu závislostí). Úkol obsahuje reference potřebné k jeho řešení (např. pro úkol rozhodnutí o zablokování IP adresy by úkol obsahoval referenci na zájmový údaj typu IP adresa).

Úkoly je taktéž možné komentovat. K řešení úkolu bez výstupu musí jeho řešitel typicky provést akci mimo „Uživatelské rozhraní“ a jeho řešení tak do „Uživatelského rozhraní“ pouze zaznamená (přidáním komentáře) a úkol uzavře. Pokud úkol obsahuje výstup, většinou je úkol nutné uzavřít zadáním hodnoty odpovídající typu výstupu.

Úkoly mohou nabývat následujících příznaků.

- Rozhodnutí (decision),
- agregátor (aggregator),
- automatizovaný (automated),
- automatická spoušť (autotrigger).

Úkol s příznakem rozhodnutí plní řešitel přímo v „Uživatelském rozhraní“ kolaborativního prostředí výběrem z jednoho z možných rozhodnutí (pravda, nepravda). Úkoly s tímto příznakem mohou sloužit k větvení (rozhodnutím pravda se otevře jiný následující úkol, než výběrem rozhodnutí nepravda).

Příznak agregátor označuje úkol, sloužící ke sjednocení výstupů z několika blokujících úkolů.

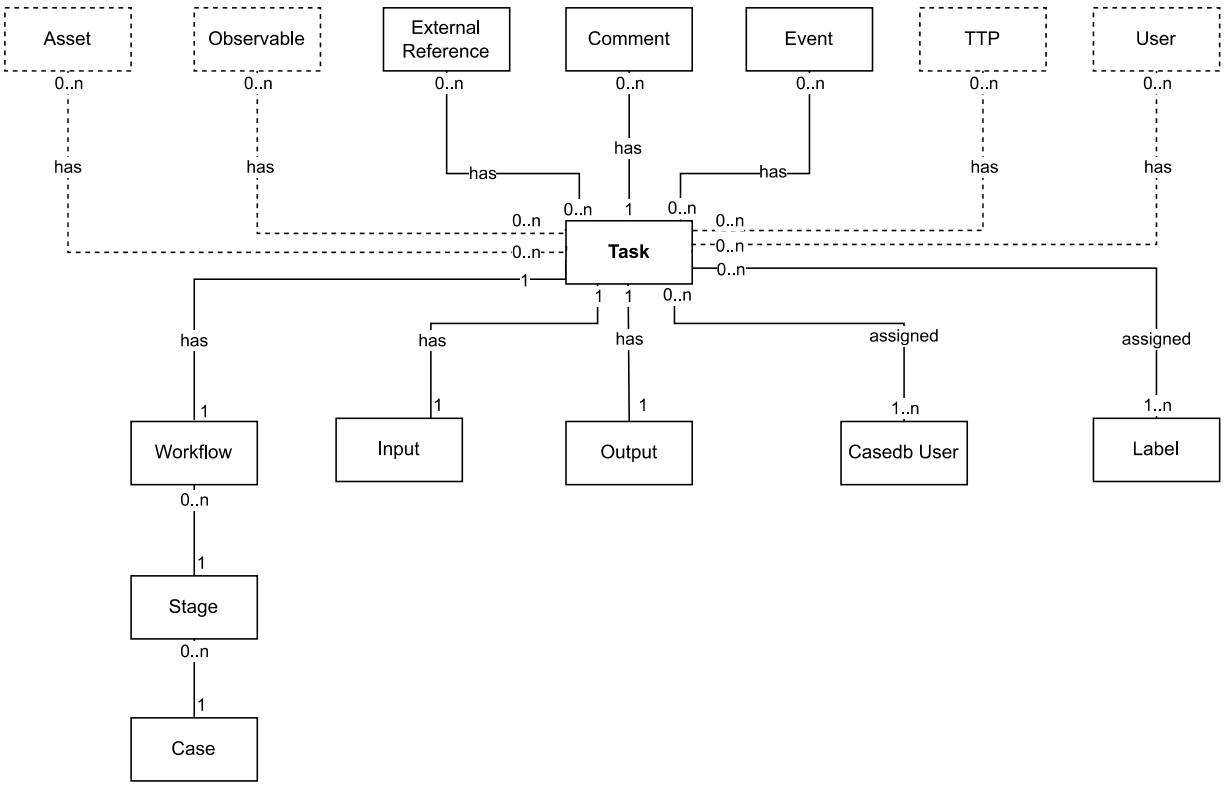
Automatizovaný příznak znamená, že bude úkol vyřešen napojeným automatizovaným procesem. Tento proces je nutné spustit tlačítkem „Trigger“. Úkolům s tímto příznakem není možné upravovat výstup v „Uživatelském rozhraní“. Stav úkolu je v pravidelných intervalech kontrolován a výstup se tak v „Uživatelském rozhraní“ zobrazí krátce po zapsání automatizovaným procesem.

Automatická spoušť se vždy vyskytuje s příznakem automatizovaný. Označuje úkoly, jejichž automatizovaný proces se spustí okamžitě po otevření úkolu. Úkol tedy nečeká na stisknutí tlačítka „Trigger“ uživatelem.

Úkoly mohou nabývat následujících stavů.

- Otevřený (open),
- čekající (pending),
- běžící (running),
- přeskočený (skipped),
- vyřešený (resolved),
- zamítnutý (rejected).

První stav nově vytvořeného úkolu je odvozen od stavu fáze pracovního postupu, do které náleží a jeho pořadí v této fázi. V případě, že se jedná o probíhající fázi, dostává první úkol stav otevřený. Všechny následující pak stav čekající. V případě, že je fáze čekající, jsou čekající i všechny úkoly. Úkol bez příznaku může být zamítnut, nebo vyřešen. Úkol s příznakem rozhodnutí může být zamítnut, nebo vyřešen s výběrem rozhodnutí (pravda / nepravda). V případě, že se jedná o automatizovaný úkol, získává úkol po startu automatizovaného procesu stav běžící. Po dokončení automatizovaného procesu se úkol dostává do stavu vyřešený. V určitých případech může automatizovaný proces nastavit stav přeskočený. Typicky se jedná o případy, kdy se rozhodnutím spustí jen určitá větev úkolů a jiná je tak přeskočena.




*Obrázek 3. Doménový model entity Úkol*

## Entity svázané s případem

Provázané entity jsou samostatné entity z interních nebo externích systémů. V rámci kolaborativního prostředí jsou charakteristické zejména svojí vazbou na úkoly.

Detailní pohled v uživatelském prostředí na konkrétní provázané entity typicky obsahuje základní informace o konkrétní entitě v režimu čtení, vazby na související případy a úkoly (pro pohyb po grafu závislostí), případně odkaz do externího systému, ve kterém je možné provádět pokročilejší operace.

### Aktivum (Asset)

Služba aktiv získává data z aplikace Netbox, která slouží k modelování infrastruktury. Původní aplikaci a její datový model rozšiřuje o CVE a vztahy s případy a úkoly.

Netbox na infrastrukturu pohlíží z pohledu tří různých domén:

- správa infrastruktury datového centra (rack, porty, zařízení),
- správa adresního prostoru (IP adresy, prefixy, služby),
- virtualizace (clustery, virtuální stroje, rozhraní virtuálních strojů).

Dále Netbox rozděluje typy aktiv rozděluje do tří kategorií:

- primární – objekty infrastruktury (IP adresa, zařízení),
- hierarchické – slouží k hierarchické organizaci aktiv (region, lokace),
- organizační – slouží ke kategorizaci (skupiny).

„Uživatelské rozhraní“ obsahuje detailní pohledy nad všemi typy aktiv. Detailní pohledy zobrazují základní informace o aktivu, jeho CVE a vazby na případy a úkoly.

### Zájmový údaj (Observable)

Model zájmových údajů vychází z formátu STIX 2.1 (Structured Threat Information Expression). Mezi typy zájmových údajů patří emailová adresa, emailová zpráva, IP adresa, URL, doména apod. Stejně jako ostatní entity této kategorie obsahuje vztahy s úkoly.

Se zájmovým údajem je spojeno také jeho CTI záznam (Cyber Threat Intelligence entry). Záznam indikuje čas, kdy byl zájmový údaj zpracován CTI nástroji a uchovává odkaz na výsledek analýzy. „Uživatelské rozhraní“ obsahuje detailní pohledy nad všemi typy zájmových údajů. Detailní pohledy zobrazují základní informace o zájmovém údaji, jeho CTI záznamy a vazby na případy a úkoly.

### TTP (Tactics, Techniques, Procedures)

Agenda TTP slouží především k budování znalostní báze taktik a technik útoků, přehled nástrojů, malwaru, či útočnických skupin. Datový model služby je inspirován modelem MITRE ATT&CK®, ale některé termíny rozlišuje jemněji a také obsahuje vazby na úkoly.

„Uživatelské rozhraní“ obsahuje detailní pohledy na všechny typy prvků znalostní báze. Detailní pohledy zobrazují základní informace o prvku znalostní báze a vztahy s dalšími prvky znalostní báze. V případě vybraných typů prvků (taktiky, techniky a pod-techniky) pak „Uživatelské rozhraní“ zobrazuje i vztahy s případy a úkoly. V postranním paneluje se nachází hierarchický strom taktik a technik.

Typy prvků znalostní báze jsou následující:

- taktika, technika (pod-technika),
- skupina a její procedury, malware a jeho procedury, mitigace,
- platforma, nástroj a jeho procedury.

### Externí reference (External reference)

Externí reference je entita odkazující do externího nástroje. Entita slouží pouze k vytvoření vztahu s úkolem. K zobrazení detailu slouží odkaz do příslušného nástroje. Reference mohou být následujících kategorií.

- Konverzace (Mattermost),
- zpravodajství (MISP),
- tiket (Gitlab, JIRA).

### Událost (Event)

Událost je obecné rozhraní popisující výskyt jevu v určitém čase s vazbou na úkol. Rozhraní může být implementováno různými specifičtějšími typy, dle charakteru události. Událost síťového toku (Flow Event) je implementací události, která k úkolu přiřazuje zaznamenané síťové toky.

### Evidovaný uživatel (User)

Evidovaný uživatel reprezentuje uživatele, který byl pozorován v rámci řešení případu. Může obsahovat vztah uživatele s IP adresou, která mu byla v určitém časovém okně přiřazena. Obsahuje také vztahy s úkolem.

V „Uživatelském rozhraní“ je možné zobrazit všechny záznamy evidovaných uživatelů, všechna přiřazení určitého uživatele různým IP adresám, nebo všechna přiřazení určité IP adresy různým uživatelům. Ke každému takovému záznamu je možné zobrazit detailní přehled síťových toků, kde daná IP adresa figuruje jako příjemce nebo odesílatel. Dále je v „Uživatelském rozhraní“ možné zobrazit vztahy s případy či úkoly.

## Back-end strana služby

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-case-database>`

Back-end strana služby a její implementace se od výše popsaných konceptů nijak zásadně neliší.

### Závislosti

- Služby:
  - Objektové úložiště kompatibilní s S3,
  - Datové úložiště Redis,
  - Relační databáze PostgreSQL,
  - Služba Uživatelské profily.
- Knihovny:
  - Knihovna DMT,
  - Knihovna pro podporu načítání konfigurace,
  - Knihovna pro podporu autentizace pomocí JWT tokenů,
  - Knihovna minio,
  - Knihovna redis.

### Implementace

Detailnější informace k implementaci back-end strany jsou v programátorském manuálu.

## 1.6. Komponenta Databáze TTP

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-ttp-database>`

Mikro-služba „*Databáze taktik, technik a procedur*“ představuje lokální úložiště známých postupů a metod používaných útočníky k dosažení svého cíle. Důvodem lokálního řešení je možnost zapojení služby do federace (potažmo integrovaného datového pohledu) a tím i možnost jednoduchého přiřazování relevantních TTP k jednotlivým případům. Sekundárním důvodem je možnost kombinovat informace z více zdrojů, případně vytvářet vlastní specifické taktiky, techniky, procedury, případně informace o nástrojích, se kterými se organizace mohla setkat.

Pro iniciální naplnění databáze je možné použít řadu open-source informačních zdrojů, z nichž asi ten nejznámější je matice MITRE ATT&CK. Pro službu byl vytvořen adaptér, který ji dokáže naplnit daty z matice MITRE ATT&CK na základě JSON dokumentu (ve standardu STIX2.1), který organizace MITRE oficiálně ke každé verzi matice vydává. V případě, že došlo k vydání nové verze matice, adaptér dokáže již uložená data aktualizovat.

### Závislosti

- Služby:
  - Relační databáze PostgreSQL.
- Knihovny:
  - Knihovna DMT,
  - Knihovna pro podporu načítání konfigurace,
  - Knihovna pro podporu autentizace pomocí JWT tokenů,
  - Knihovna mitreattack-python.

### Implementace

Služba nabízí GraphQL API, které bylo z velké většiny vygenerované pomocí meta-programovacího mikro-frameworku DMT [3]. Výsledné schéma bylo rozšířeno o dvě mutace, které slouží k vytváření entity Technika ve vztahu „Pod-technika“.

Adaptér je z pohledu implementace složitější. Jako zdroj dat používá oficiální soubory JSON, které organizace MITRE zveřejňuje ke každé verzi matice. Popis matice je vytvořen ve standardizovaném formátu STIX 2.1.

Adaptér si při prvním spuštění vytvoří konfigurační soubor, ve kterém následně udržuje poslední verzi matice, kterou se mu podařilo do databáze nahrát. Adaptér je možné spustit ve dvou režimech: režim aktualizace dat a režim doplnění dat. Režim doplnění dat při případných změnách nahrává pouze ty taktiky, techniky a procedury, které jsou v MITRE matici nové. Režim aktualizace dat kromě nahrávání nových TTP aktualizuje i již známé TTP, u kterých došlo v matici ke změnám.

## 1.7. Komponenta Uživatelské profily

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-user-profile-introspection-service>`

Mikro-služba uživatelské profily představuje GraphQL API umožňující vyhledávat a získávat informace o uživatelích uložených ve vybraném nástroji poskytovatele identit (Keycloak). Tyto informace nejsou součástí speciálního JWT tokenu, který posílá služba *„Servisní brána“* na zabezpečené mikro-služby, neboť *„Servisní brána“* z principu nepracuje s pokročilými informacemi o uživatelích, tj. neplní funkci poskytovatele identit.

Pokud nějaká komponenta vyžaduje pro svoje fungování bližší informace o uživateli, musí k nim přistoupit pomocí této mikro-služby. Typickými službami, které tuto službu využívají, jsou *„Databáze případů“*, pro vyplnění interního uživatelského profilu (jméno, e-mailová adresa) a služba *„Databáze uživatelské aktivity“*, pro umožnění introspekce nad evidovaným uživatelem.

### Závislosti

- Služby:
  - Poskytovatel identit Keycloak.
- Knihovny:
  - Knihovna pro podporu načítání konfigurace,
  - Knihovna pro podporu autentizace pomocí JWT tokenů.

### Implementace

Pro dotazování nad Keycloak API je implementována klientská třída. Klient umožňuje vyhledávání uživatelů například dle e-mailových adres, uživatelských jmen či aliasů. Dále lze pomocí klienta získat detaily o uživateli s daným ID.

GraphQL API poskytuje rozhraní pro získání detailů o uživateli z autentizačního tokenu, či pomocí pevně daného uživatelského ID a tzv. *realmu*. Pomocí dotazu *user_search* lze také vyhledávat uživatele.

## 1.8. Komponenta Servisní brána

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-service-gateway>`

Mikro-služba „Servisní brána“ představuje jednotný vstupní bod zaštiťující všechny ostatní mikro-služby projektu. Jde o netriviální mikro-službu, která splňuje velmi vysoké nároky s ohledem na zabezpečení a výkonnost. Služba implementuje autentizační a autorizační mechanismus, který umožňuje přenos režie vyvstávající z použití protokolu OpenID Connect (OIDC) z mikro-služeb na centrální službu, čímž je dosaženo výrazného zjednodušení bezpečnostních mechanismů na ostatních mikro-službách, při zachování bezpečnosti a granularity protokolu OIDC.

OIDC je otevřený autentizační protokol, založený na standardu OAuth2, jehož cílem je poskytnout dalším službám nástroj k ověření identity uživatele a získání základních informací o uživateli. Používá JSON Web Tokeny (JWT), ve kterých služba využívající OIDC obdrží údaje o uživateli a jeho vlastnostech (privilegia, členství v organizačních jednotkách apod.). Z pohledu uživatele se jedná o tzv. jednotné přihlášení (například „Single Sign On“). Pro realizaci tzv. poskytovatele identit byl zvolen nástroj Keycloak.

Koncept centrální vstupní brány znamená, že služba je schopná vystupovat jako jednotný vstupní bod pro aplikační rozhraní všech mikro-služeb, které za ní leží. V praxi brána představuje reverzní proxy server, schopný jako prostředník zastupovat vyvinuté mikro-služby. Výběr dotazované služby probíhá pomocí detekce identifikujícího URL segmentu (například *…/muni*, *…/csirt*) na bázi rozhodovacích pravidel, která jsou definována v konfiguračním souboru mikro-služby.

Centrální brána před přeposláním originálního dotazu na žádoucí službu provádí vůči poskytovateli identit autentizaci. Při úspěšné autentizaci brána vytvoří speciální JWT token, kterým nahrazuje původní token získaný od uživatele.

Speciální token generovaný službou umožňuje standardizaci autentizačních a autorizačních mechanismů bez ohledu na drobné odlišnosti v tokenech mezi konkrétními poskytovateli identit. Použití speciálního tokenu také v konečném důsledku celý mechanismus urychluje, neboť se ukazuje, že „nejdražší“ operací je komunikace s poskytovatelem identit. V případě speciálních tokenů s jednotným přihlášením komunikuje pouze centrální služba, generované tokeny je možné ukládat do rychlého úložiště sloužícího jako mezipaměť. Jednotlivé mikro-služby proto s poskytovatelem identit vůbec nekomunikují, důvěřují tokenům odeslaným z centrální brány. Důvěra v tokeny je podpořena mechanismem symetrického šifrování.

### Závislosti

- Služby:
  - Relační databáze PostgreSQL,
  - Poskytovatel identit Keycloak,
  - Datové úložiště Redis.
- Knihovny:
  - Knihovna pro podporu načítání konfigurace [1],
  - Knihovna pomocných autentizačních funkcí [1],
  - Knihovna aioredis.

### Implementace

Implementace služby je detailněji popsána v programátorském manuálu.

## 1.9. Komponenta Uživatelské rozhraní

**Cesta v adresáři výsledku:** `<frontend-angular/csirtmu-orion-frontend>`

„Uživatelské rozhraní“ má podobu webové aplikace ve frameworku Angular a jejím základem je sada komponent Angular Components. Pokročilá funkcionalita a komponenty jsou implementovány pomocí mikro-frameworků Sentinel a ORION SCMFM. „Uživatelské rozhraní“ je rozděleno do několika agend. Agendy jsou logické celky spojující vzájemně provázané součásti aplikace. Každé agendě typicky náleží příslušná mikro-služba na straně back-end.

Webová aplikace „Uživatelského rozhraní“ se skládá z následujících agend.

- Aktiva (assets),
- zájmové údaje (observables),
- IPFIX,
- evidence uživatelů (user accounting),
- případy (cases),
- štítky (labels),
- šablony pracovního postupu (workflow templates),
- přehledová obrazovka (dashboard),
- TTP (tactics, techniques, procedures).

Jednotlivé agendy, vyjma specializované agendy dashboard, můžeme dále rozdělit do několika kategorií.

- Přehledové (aktiva, zájmové údaje),
- monitoring (IPFIX, evidence uživatelů),
- správa případů (případy, štítky, šablony pracovního postupu),
- znalostní báze (TTP).

### Angular

Angular framework je open-source modulární webový framework, který je možné použít k budování rozšiřitelných webových aplikací a knihoven. Angular využívá programovací jazyk TypeScript, což je open-source rozšíření JavaScriptu o typový systém a statickou typovou kontrolu. Jelikož je TypeScript kompilován zpět do JavaScriptu, je podporován všemi moderními webovými prohlížeči. Angular je postaven na komponentové architektuře a umožňuje tak skládat komplexní uživatelské rozhraní z malých, znovupoužitelných a lehce udržovatelných dílů.

Každá komponenta je složena z anotované třídy navázané na HTML šablonu a CSS soubor. Komponenty využívají standardní HTML rozšířený o vlastní značky zvané direktivy. Skládáním komponent vzniká v aplikaci jejich hierarchický strom. Angular umožňuje jednoduchou komunikaci pomocí předávání dat či vyvolávání událostí mezi dvěma komponentami v přímém hierarchickém vztahu. Pro komunikaci napříč větším množství komponent je možné využít tzv. služby.

V kontextu Angularu je službou jakákoliv specificky anotovaná třída, kterou framework rozpozná a vytvoří její instanci. Tuto instanci následně poskytuje všem komponentám či dalším službám, které jsou na ni závislé. Jedná se o implementaci principu vkládání závislostí. Komponenty i služby jsou seskupeny do modulů, které tvoří jejich logický rámec. Webové aplikace většího rozsahu tak jsou sestavovány vytvářením vlastních modulů a rozšiřovány importováním modulů z externích knihoven.

Za předpokladu dodržení určitých architektonických principů a doporučených postupů, jsou aplikace a knihovny vyvinuté v Angularu vysoce modulární, znovupoužitelné a lehce rozšiřitelné. Mimo prostředí pro tvorbu komponent, poskytuje Angular mnoho užitečných nástrojů pro směrování mezi jednotlivými pohledy aplikace, HTTP komunikaci nebo tvorbu animací. Angular je dodáván s integrovanými testovacími nástroji a sám poskytuje komplexní aplikační rozhraní pro testování komponent a služeb.

## GraphQL

GraphQL je dotazovací jazyk, umožňující klientské aplikaci flexibilně a přesně popsat své datové požadavky a interakce. GraphQL služba se skládá z typů a polí definovaných na těchto typech. Specifikace GraphQL obsahuje základní sadu typů, ale služba může typový systém rozšiřovat. Nad definovanými typy je možné provádět operace (dotazy a mutace). Operace mohou očekávat vstupní argumenty. Operace lze vnořovat, a i vnořené operace mohou přijímat vstupní argumenty. Dotazy (queries) jsou operace, které jsou vykonávány paralelně a typicky jsou používány pro získávaní dat bez jejich modifikace. Mutace (mutations) jsou operace, které jsou prováděny sekvenčně a typicky jsou používány k vytváření, modifikaci a odstranění existujících dat. K vytváření dynamických operací lze využít proměnné, které jsou do dotazu vloženy z externího zdroje datového typu slovník. Pro zvýšení znovu-použitelnosti jednotlivých částí dotazu lze použít fragmenty. Fragment je množina vybraných polí určitého typu, kterou je možno opakovaně odkazovat z jedné či více operací.

GraphQL server poskytuje staticky typované schéma, validuje a vyhodnocuje příchozí dotazy a mutace. Klientská aplikace tedy může vytvořit dotaz, který přesně popisuje data potřebná např. k zobrazení dané obrazovky. Za předpokladu vhodného návrhu GraphQL schéma, dochází v porovnání s REST API k dramatické redukci množství komunikace i objemu jednotlivých zpráv (klient dokáže spojit několik dotazů do jednoho HTTP požadavku a v odpovědi dostává pouze ta data, o která žádal). Použití GraphQL také umožňuje snadnější rozšiřování a změny aplikačního rozhraní, bez nutnosti verzování a integrace případných změn.

## Apollo Client

Apollo Client je JavaScript knihovna pro správu stavu webové aplikace. Knihovna používá GraphQL k řízení stavu lokálních i vzdálených dat. Umožňuje získávání dat, modifikaci dat a aktualizace stavu uživatelského rozhraní. Mezi pokročilé funkce knihovny patří ukládání do mezi-paměti, komplexní správa mezi-paměti, podpora stránkování, detekce chyb, reakce na chyby a HTTP komunikace.

## Apollo Angular

Apollo Angular je knihovna integrující knihovnu Apollo Client se zobrazovací vrstvou Angular webové aplikace. Knihovna obaluje funkcionalitu Apollo Client do aplikačního rozhraní snáze použitelného v prostředí frameworku Angular, poskytuje konfigurovatelný modul a poskytuje dotazy a mutace formou Angular služeb.

## GraphQL Code Generator

Nástroj GraphQL Code Generator dokáže generovat kód pro back-end i front-end aplikace z GraphQL schémat a operací. Generátor je rozšiřitelný o celou řadou pluginů pro konkrétní programovací jazyky či knihovny. Ve front-end části aplikace kolaborativního prostředí je nástroj využíván ke generování TypeScript typů a služeb z GraphQL schématu.

## Sentinel CSIRT-MU

Sentinel je otevřený mikro-framework pro tvorbu front-end rozhraní týmu CSIRT-MU, který rozšiřuje framework Angular a sadu komponent Angular Components. Framework se skládá se ze čtyř knihoven, které tvoří vzájemně se doplňující sadu nástrojů usnadňující a urychlující vývoj webových aplikací. Tyto knihovny zajišťují jednotný vzhled a chování aplikací týmu CSIRT-MU a pro potřeby projektu museli být patřičně rozšířeny pro kompatibilitu s frameworkem ORION SCMFM. Konkrétně se jedná o:

- Sentinel Common,
- Sentinel Layout,
- Sentinel Auth,
- Sentinel Components.

Balík Sentinel Common obsahuje zejména třídy, rozhraní a funkce implementující základní funkcionalitu, která se opakuje v mnoha webových aplikacích. Ku příkladu se jedná o stránkování, třídy zjednodušující udržování stavu aplikace a další utility. Nejdůležitější součástí tohoto balíku je skupina tříd poskytující modifikovanou sekvenci startu Angular aplikace, která umožňuje získat konfigurační soubor ze serveru. Toto řešení má oproti výchozímu startu Angular aplikací mnoho výhod, jednou z ních je podpora kontejnerizace.

Sentinel Layout obsahuje několik komponent tvořící sdílené rozvržení grafických elementů typické aplikace. Rozvržení se skládá z postranního panelu obsahující navigaci mezi jednotlivými agendami, panelu nástrojů, profilu uživatele a uživatelského menu, kontextového postranního panelu, drobečkovou navigaci a notifikací. Vývojář aplikace tedy vyvíjí jen samostatné obrazovky, které jsou do vybraného rozložení vkládány.

Sentinel Auth je knihovna pro jednoduchou integraci autentizace a autorizace. Obsahuje sadu komponent a tříd implementující proces přihlášení a odhlášení. Autentizační část umožňuje výběr z několika OIDC poskytovatelů a následné přihlášení jehož prostřednictvím. K procesu komunikace s OIDC poskytovatelem a získání autentizačního tokenu je využita externí knihovna *angular-oauth2-oidc*, která je aktivně udržována a certifikována neziskovou organizací OpenID Foundation. Autorizaci knihovna přímo neprovádí, jelikož tato část přihlašovacího procesu je většinou specifická pro danou aplikaci. Knihovna ovšem umožňuje poskytnout vlastní implementaci, dle návrhového vzoru strategie.

Poslední součástí mikro-frameworku Sentinel je balík Sentinel Components, poskytující celkem 15 znovupoužitelných komponent (tabulka, dialogy, nástrojový panel atd.)

## Mikro-framework ORION SCMFM

ORION SCMFM je mikro-framework postavený na frameworku Angular, knihovně Angular Components a sadě knihoven Sentinel. Jeho účelem je identifikovat a abstrahovat opakující se vzory v „Uživatelském rozhraní“, dle analýzy požadavků na „Uživatelské rozhraní“ a tyto vzory implementovat formou generických komponent a služeb. Mikro-framework také poskytuje několik znovupoužitelných komponent, které jsou v „Uživatelském rozhraní“ hojně využity.

Mezi abstrahované vzory patří služby poskytující implementaci stránkování, řazení a filtrování ve variantách uložení stavu do paměti aplikace a uložení stavu do dotazové části URL. Dále se jedná o služby využívající Apollo ke komunikaci s GraphQL API a jejich varianty podporující stránkování, řazení a filtrování. Na míru variantám těchto služeb jsou implementovány generické třídy, které mohou rozšiřovat chytré komponenty komunikující se službami Apollo a případně obsahující ovládací prvky pro změnu stránkování, řazení, nebo filtrování.

Podrobnější popis Mikro-frameworku ORION SCMFM se nachází níže, v sekci „Mikro-framework ORION SCMFM“ Programátorského manuálu.

# 2. Programátorský manuál

## 2.1. Komponenta Servisní brána

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-service-gateway>`

### Reverzní proxy

Reverzní proxy je implementována pomocí dvou knihoven, první je Starlette, minimalistický ASGI (*Asynchronous Server Gateway Interface* – standardizovaný Python interface pro asynchronní webové frameworky, respektive servery a aplikace). Starlette poskytuje minimální základ pro webový server a zároveň, díky velmi dobré rozšiřitelnosti, lze nad ní snadně implementovat všechny potřebné rozšíření, například vlastní autorizaci a autentizaci. Jednou z funkcionalit, kterou poskytuje je tzv. routing, tedy schopnost rozhodovat o způsobu zpracování požadavku dle definovaných pravidel, v našem případě na základě sekce URL adresy.

Každá mikro-služba je identifikována unikátním segmentem URL, ke kterému je přiřazená cílová adresa a schéma (HTTP/S) dané mikro-služby. Část URL za identifikátorem se považuje za URL cestu určenou pro cílovou aplikaci, HTTP metoda je děděna z původního dotazu.

Po sestavení požadavku pro cílovou mikro-službu ho „*Servisní brána*“ sama odešle pomocí druhé klientské knihovny AIOHTTP, která zajišťuje asynchronní HTTP/S požadavky. Asynchronní implementace klientské knihovny i samotného web serveru, který tvoří jádro reverzní proxy, umožňují, aby mohla služba pracovat na více požadavcích zároveň – zatímco jeden požadavek čeká na odpověď od cílové mikro-služby, může brána zpracovávat ostatní paralelně běžící požadavky.

Paralelizace umožňuje prakticky eliminovat blokování kódu čekáním na vstupně / výstupní operace, které jsou principiálně časově nejnáročnější operací u proxy serverů – čekání na odpověď od dotazované služby tvoří až 99 % nákladu výpočetního času. Odbouráním režie vstupně / výstupních operací a designem, který je schopný horizontálně i vertikálně škálovat, bylo dosaženo potřebného výkonu pro to, aby služba mohla sloužit jako vstupní bod pro prakticky libovolný počet mikro-služeb.

### OIDC Akcelerace a Caching

Druhou rolí této služby je na sebe převzít odpovědnost za komunikaci s poskytovatelem identit. Komunikace s poskytovatelem identit byla identifikována jako druhá časově nejnáročnější operace, navíc se v případě OIDC nejedná o implementačně triviální proces, který při každém nasazení vyžaduje pro dané prostředí specifickou konfiguraci (bezpečnostní klíče, identifikátory klientů apod.).

Z těchto důvodu „*Servisní brána*“ představuje jediného OIDC klienta (klient je z pohledu OIDC služba, která OIDC využívá pro autentizaci uživatelů).

Služby za branou od ní dostávají speciální JWT tokeny. JSON Web Token je formát pro předávání informací mezi dvěma stranami, zajišťující integritu obsahu pomocí podpisů, volitelně je schopný zajistit i důvěrnost pomocí šifrování.

Pro účely popisu fungování autentizačního procesu je z pohledu tokenů důležité vysvětlit koncept tzv. claims (dále překládáno jako *tvrzení*). Jedná se o páry klíč a hodnota, kde klíč je jakýkoliv textový řetězec a hodnota jakýkoliv validní JSON řetězec. Tyto tvrzení jsou do těla tokenu přidávána na základě tzv. scopes, což je logické označení pro skupinu tvrzení.

Autentizovaná komunikace s mikro-službami za „*Servisní branou*“ probíhá následovně:

1. Uživatel získá od OIDC poskytovatele svůj JWT autorizační token.
2. Pošle dotaz na „*Servisní bránu*“, který obsahuje hlavičky s tokenem a adresu OIDC poskytovatele, od kterého získal token.
3. Pokud není v nastavení „*Servisní brány*“ určen výchozí OIDC poskytovatel, musí být specifikován v hlavičce každého požadavku, protože brána je schopná zastupovat tokeny od více poskytovatelů.

4. „*Servisní brána*“ ověří token u specifikovaného poskytovatele, extrahuje tvrzení, ze kterých vybere ty relevantní, tzv. sledovaná tvrzení. Sledovaná tvrzení jsou součástí skupiny tvrzení, která slouží pro autentizaci uživatelů a „*Servisní brána*“ je bude předávat mikro-službám.
5. „*Servisní brána*“ následně vytvoří svůj vlastní JWT token, podobný tokenům získaných od OIDC. Jedná se o zjednodušenou verzi, obsahující pouze informace relevantní pro autorizaci a autentizaci na koncových mikro-službách. V tomto zjednodušeném tokenu, vystupuje „*Servisní brána*“ jako „issuer“ tedy OIDC poskytovatel, který token vydal. Token obsahuje následující položky:
    a. *username*: unikátní identifikátor uživatele napříč všemi OIDC poskytovateli, tvořený adresou poskytovatele a identifikátorem uživatele unikátním v rámci daného vydavatele. Tato kombinace zajišťuje unikátnost uživatelských identifikátorů napříč vícero OIDC poskytovateli.
    b. *scope*: list hodnot, které se používají pro specifikování přístupových privilegií, o které klient žádá.
    c. *jti*: unikátní identifikátor tokenu.
    d. *exp*: datum a čas expirace tokenu (konec platnosti), údaj je děděný z původního tokenu.
    e. *iat*: datum a čas vydání tokenu.
    f. *iss:* issuer – identifkátor OIDC poskytovatele.
    g. *aud*: audience, pro koho (OIDC autentizační server, klienta, službu) je token určený.
    h. capabilities – seznam „hodnot“ extrahovaných tvrzení (klíče se nepředávají).
6. Token následně podepíše svým symetrickým klíčem, který jí byl nastaven, a je stejný mezi všemi mikro-službami. Mikro-služby klíč používají k ověření integrity tokenu.
7. Token připojí k dotazu, který přeposílá na cílovou mikro-službu.

Vlastní tokeny (tzn. vydané službou) jsou využívány kvůli již zmíněné extrakci a sjednocení relevantních tvrzení do „capabilities“. Dále umožňují přidávat vlastní tvrzení a do značné míry sjednotit formát těla tokenu.

Nejvýznamnějším benefitem zpracovávání a verifikace OIDC tokenů již na této službě je možnost zpracované tokeny cachovat do velmi rychlé databáze Redis typu klíč-hodnota. Tokeny jsou cachovány po jejich prvním ověření. Jako klíč slouží původní token od OIDC poskytovatele a token z něj vytvořený službou je uložen jako „hodnota“. Díky tomu může „*Servisní brána*“ při každém zpracovávaném požadavku velmi rychle zkontrolovat, zdali zpracovávaný token již není uložen v cache paměti, pokud je, nemusí jej znovu ověřovat u OIDC poskytovatele ani dále zpracovávat, pouze nahraje hodnotu z cache. Akceptaci expirovaného tokenu je znemožněno děděním jeho expirace na expiraci záznamu v cachi.

Posledním, ale ne méně významným benefitem vlastních tokenů je zvýšení rychlosti zpracování tokenu na koncových mikro-službách při zachování bezpečnosti, a to použitím symetrického šifrování, respektive podepisování tokenů. Vzhledem k faktu, že koncové mikro-služby nemusí tokeny ověřovat u OIDC poskytovatele, protože díky znalosti podpisového klíče, zvládnou ověření provést lokálně. Díky tomu koncové mikro-služby nepotřebují vůbec komunikovat s OIDC poskytovateli, což lze také samo o sobě považovat za benefit z pohledu bezpečnosti.

## API Tokeny

API tokeny nebo také servisní tokeny jsou další funkcí „*Servisní brány*“, jedná se o speciální typ tokenu, který není spojený s uživatelem, respektive účtem / identitou u OIDC poskytovatele. Servisní tokeny slouží pro připojení služeb třetích stran, které nepodporují OIDC, nebo ho není možné použít. Expirace servisních tokenů je oproti OIDC autentizačním tokenům podstatně delší (v řádech desítek dnů až měsíců – tato hodnota je konfigurovatelná). Obsah servisního tokenu kopíruje standardní OIDC autentizační tokeny, „claims“ je konfigurovatelná položka, uživatel si při vytváření tokenu může vybrat potřebné tvrzení z předem definovaného seznamu (ten je součástí konfigurace mikro-služby). Tokeny jsou ukládány v relační databázi.

## Horizontální a vertikální škálování

Horizontální škálování je možné a efektivní díky sdílené Redis cachi, která sama podporuje klusterování. Vertikální škálování realizujeme pomocí tzv. „workerů“, aplikace je schopná se rozdělit do několika procesů („workerů“), mezi které se rozdělí zpracovávané požadavky, což umožňuje efektivně využít všechna dostupná procesorová jádra.

## 2.2. Knihovna pro podporu autentizace pomocí JWT tokenů

**Cesta v adresáři výsledku:** `<backend-python/libraries/csirtmu-starlette-jwt-auth-backend>`

Pro použití autentizace v aplikacích pomocí služby „*Servisní brána*“ byla vytvořena tato knihovna pro zpracování přeposlaných autorizačních tokenů v aplikaci. Cílem je poskytnout abstrahované detaily o uživateli a jeho oprávněních z důvodu jednoduché a přímočaré konfigurace autorizace. Knihovna je navržena pro spolupráci s ASGI frameworkem Starlette.

Při využití této knihovny je aplikován vzor „*middleware*“, při kterém je kód používán jako prostředník mezi dvěma jinými částmi kódu. V tomto případě middleware předává příchozí dotaz po úrovních až do samotného aplikačního kódu, přičemž tato knihovna je použita jako autentizační middleware.

Knihovna se skládá z několika hlavních částí:

### TokenAuthenticationBackend

Hlavní třída poskytovaná knihovnou. Používá se v autentizačním middleware pro dekódování informací získaných z JWT tokenů do objektů uživatele. Během autentizace kontroluje třída validnost tokenu na základě předkonfigurovaných kritérií a přiřazuje vytvořeným uživatelským objektům práva.

```
app.add_middleware(AuthenticationMiddleware,
        backend=TokenAuthenticationBackend())
```

*Obrázek 4. Připojení autentizačního middleware k aplikaci*

### Konfigurace

Knihovnu lze konfigurovat stejným způsobem jako mikro-služby, tedy pomocí YAML souboru či proměnných prostředí. Pevně dané schéma konfigurace se nachází v *config* modulu a nastavuje bezpečnostní parametry tokenů jako iss, aud, key apod.

#### Třída uživatele

Objekty uživatelů jsou vytvářeny z třídy *MappedUser*, dědící z *BaseUser* třídy ze Starlette knihovny. Uživatelé mají přidanou statickou metodu *create_or_get*, kterou může aplikace přepsat a umožnit mapování na databázové modely.

```
class UserMeta(type(Base), type(MappedUser)):
    pass


class DatabaseUser(Base, MappedUser, metaclass=UserMeta):

TokenAuthenticationBackend(user_class=DatabaseUser))
```

*Obrázek 5. Třída uživatele*

#### Koncový bod pro zajištění kompatibility

V aplikaci lze využít koncový bod pro zajištění kompatibility, který odpovídá na požadavky s aktuální verzí nainstalované knihovny. Pro zajištění kompatibility rozhraní pak služba „*Servisní brána*“ tuto verzi kontroluje se svou tabulkou kompatibility a může varovat před konflikty.

### Konfigurace autorizačních pravidel

Pro autorizační pravidla je využívaná knihovna *Oso* a její formát souborů *polar.* V *polar* souborech jsou definovány takzvané zdroje, subjekty a pravidla. Pravidla se vždy aplikují na subjekt, zdroj a nějaký název (např. role, akce apod.).

Schopnosti uživatelů bývají namapovány na role pomocí pravidla:

```
has_role(actor: User, role: String, _resource: KeycloakClient) if
    role in actor.capabilities;
```

*Obrázek 6. Uživatel v souboru polar*

Přístup a oprávnění u jednotlivých zdrojů je pak řízen těmito skupinami:

```
resource KeycloakClient {
    permissions = [ "can:query", "can:search", "can:self-introspect" ];
    roles = [ "is:default" ];

    "can:query" if "is:default";
    "can:search" if "is:default";
}

has_permission(_actor: User, "can:self-introspect", _resource: KeycloakClient);

allow(actor, action, resource) if
    has_permission(actor, action, resource);

actor User {}
```

*Obrázek 7. Aplikace oprávnění v souboru polar*

Ačkoli je typicky používán tento způsob autorizace, dovolují *polar* soubory vysokou flexibilitu, granularitu a kontrolu nad oprávněními.

## 2.3. Knihovna pomocných autentizačních funkcí

**Cesta v adresáři výsledku:** `<backend-python/libraries/csirtmu-auth-utils>`

Tato knihovna obsahuje pomocné funkce, používané ve službách primárně pro zpracování JWT tokenů. Aktuálně je využívaná tranzitivně, skrze knihovnu pro podporu autentizace pomocí JWT tokenů (viz předchozí sekce), ale lze ji použít i samostatně.

### Autentizační mezipaměť (Authentication Cache)

Třída AuthenticationCache se nachází v modulu *authentication_cache* a poskytuje funkcionalitu mezipaměti s omezenou kapacitou, mazáním nejdéle nepoužitého záznamu a omezenou životností záznamu.

Poskytované rozhraní nabízí metody *add* pro přidání hodnoty pod klíčem do mezipaměti s možností specifikace životnosti a *get*, jež pro daný klíč získá uloženou hodnotu.

### OIDC Issuer

Další třídou, nacházející se v modulu *oidc_issuer*, je OpenIDConnectIssuer. Tato třída je používána k zasílání požadavků na adresu nějakého OIDC poskytovatele. Konkrétně se jedná o získání informací z *userinfo* koncového bodu, ddekódování a verifikace JWT tokenu, získání metadat o poskytovateli a načtení JWK klíčů poskytovatele.

### Utils

V posledním modulu *utils* se nachází tři pomocné funkce:

- **extract_access_token –** z hodnoty autorizační HTTP hlavičky extrahuje přístupový JWT token.
- **extract_auth_header** – vrátí hodnotu autorizační http hlavičky (standardně „Authorization“).
- **collect_watched_claims** – z dekódovaných dat JWT tokenu vrátí seznam „capabilities“, tedy zakódovaných práv daných hodnotami sledovaných „claims“ polí.

## 2.4. Komponenta Databáze případů

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-case-database>`

Mikro-služba nabízí GraphQL API. Základní CRUD operace a velmi pokročilé možnosti filtrování u *read* operací jsou automaticky generované meta-programovací knihovnou DMT.

Do služby bylo nicméně doimplementováno poměrně velké množství vlastních mutací, které v sobě mají zabudovanou složitější než jen základní logiku. Některé z těchto mutací dokonce nahrazují i mutace vygenerované knihovnou DMT.

Implementaci služby je možné rozdělit do následujících vrstev.

### Databázová vrstva

Model služby je definován stejným způsobem, jako v ostatních službách používajících knihovnu DMT.

Databázový model je obohacen také o několik triggerů (ty jsou implementovány pomocí migračního nástroje *Alembic*), které mají za úkol dynamicky nastavovat stavy pracovních postupů a fází pracovních postupů podle úprav stavů úkolů, které se v nich nacházejí. Původní variantou bylo řešení pomocí konceptu *column_property* v knihovně *SQLAlchemy*, který umožňuje načítat hodnotu atributu dynamicky i s použitím SQL dotazu. Toto řešení bylo nicméně velmi pomalé, jelikož například pro výpis pracovního postupu načítala knihovna *SQLAlchemy* jednotlivé stavy úkolů sekvenčními dotazy, což při rozsáhlejším postupu generovalo velké množství dotazů. Při použití triggerů pro nastavování hodnot stavů generování dynamických dotazů odpadlo. Ke spouštění triggeru zároveň dochází pouze v případě, kdy se mění nebo přidává úkol, což je méně často než v případě čtení.

### Servisní vrstva

Servisní vrstva se skládá z několika komponent: implementace přídavných mutací, práce se šablonami úkolů a pracovních postupů, práce se soubory, generování uživatelských ikon, introspekce uživatelských účtů z nástroje poskytovatele identity.

#### Implementace přídavných mutací

Nejdůležitější komponentou je implementace přídavných mutací. Přídavné mutace realizují složitější logiku služby a řeší komplikovanější vazby. Přídavné mutace implementují například následující funkcionalitu.

- Uzavírání úkolů a zamítání úkolů: implementace logiky kontroly podmínek, otevírání a přeskakování navazujících úkolů, předávání vstupu a výstupu apod., je řešeno i několik typů mutací pro uzavření úkolu (běžné uzavření, uzavření s rozhodnutím TRUE / FALSE),
- vytváření úkolů: implementace logiky vytváření vztahů, přípravy vstupu úkolu, předávání hodnoty z výstupu rodiče, aplikace flagů,
- vytváření případů: především je řešena možnost zadávání identifikátoru šablony pracovního postupu, který chceme zároveň s případem vytvořit a do případu přidat, dále i přípravy kořenového adresáře případu pro vkládání souborů,
- logiku blokovaných a blokačních úkolů,
- agregace výstupů,
- vytváření souborů,
- vytváření souborů z komentáře.

#### Komponenta pro práci se šablonami

Komponenta pro práci se šablonami má následující využití.

- Možnost automatického vygenerování šablony na základě existujícího pracovního postupu nebo úkolu,

- vytvoření pracovního postupu nebo úkolu z existující šablony,
- možnost exportu šablony do předpisu ve formátu JSON,
- možnost importu šablony z předpisu ve formátu JSON.

Šablony samotné jsou popsány v závěru sekce.

### Komponenta pro práci se soubory

Komponenta pro práci se soubory slouží k ukládání uživatelem nahraných souborů na objektové úložiště S3 a k zpřístupňování těchto souborů prostřednictvím speciálního endpointu. Práce se soubory vypadá následovně.

1. Uživatel nahraje soubor do služby.
2. Služba nahraje soubor do objektového úložiště.
3. Služba si vytvoří v databázi entitu *File*, na kterou je možné se dotazovat prostřednictvím GraphQL API. Entita mj. obsahuje atribut *src*, který obsahuje URL odkazující na speciální endpoint služby a cestu k souboru.
4. Uživatel položí dotaz a nechá si vrátit entitu *File*.
5. Uživatel přistoupí na cestu odkazovanou atributem *src.*
6. Služba z cesty dekóduje, o který soubor v objektovém úložišti se jedná.
7. Služba vrátí obsah souboru.

### Komponenta generování ikon

Generování uživatelských ikon je jednoduchá komponenta, která slouží pro generování výchozích ikon uživatelů.

### Introspekce uživatelů

Introspekce uživatelských účtů je poměrně důležitou komponentou, která slouží k získávání dodatečných informací o uživateli ze služby „*Uživatelské profily*“.

### GraphQL schéma

GraphQL schéma je z větší části vygenerováno na základě modelu, nicméně je doplněno i o poměrně velké množství vlastních mutací. Většina těchto mutací má svůj protějšek v servisní vrstvě, v jejíž části byly i do jisté míry popsány.

## Šablony

Vrchní úroveň předpisu ve formátu JSON je pracovní postup. Na dále uvedené šabloně (Obrázek 8) můžeme vidět položky jméno a popis (ten byl z důvodu úspory místa zkrácen). Předpis pracovního postupu dále obsahuje položku *stages*, která představuje jednotlivé fáze. U fází, podobně jako u pracovních postupů, vyplňujeme název a popis. Fáze se dále člení do úkolů (*tasks*).

```
{
  "name": "airflow-phishing-template",
 "description": "...",
 "stages": [{"name": "Analysis Phase",
         "description": "Analyse machine confirmed phishing",
         "tasks": [{ "id": 1,
            "name": "Confirm phishing",
            "description": "...",
            "is_decision": true,
            "on_true": [2, 4, 6, 3, 5]
            },
            {"id": 2,
            "name": "Aggregate malicious domains",
            "description": "...",
            "input_type": "json",
            "output_type": "json",
            "on_completion": [8],
            "aggregates_results": true
            },
            {"id": 3,
            "name": "Persist confirmed phishing",
            "description": "...",
            "input_type": "json",
            "is_automated": true,
            "autotrigger": true,
            "processing_queue": "persist_phishing_queue"
            },
            {"id": 4,
            "name": "Aggregate malicious IP addresses",
            "description": "...",
            "input_type": "json",
            "output_type": "json",
            "on_completion": [7],
            "aggregates_results": true
            },
            {"id": 5,
            "name": "Aggregate malicious email addresses",
            "description": "...",
            "input_type": "json",
            "output_type": "json",
            "on_completion": [9],
            "aggregates_results": false
            },
            {"id": 6,
            "name": "Aggregate malicious files",
            "description": "...",
            "input_type": "json",
            "output_type": "json",
            "on_completion": [10],
            "aggregates_results": true
            }]
         },
         {"name": "Mitigation Phase",
          "description": "...",
          "tasks": [{"id": 7,
            "name": "Blacklist malicious IP addresses",
            "description": "...",
            "input_type": "json",
            "input_format": "json schema",
            "output_type": "json",
            "processing_queue": "exafs_queue",
            "is_automated": true,
            "autotrigger": true,
            "on_completion": [10]
            },
            {"id": 8,
            "name": "Blacklist malicious domains",
            "description": "...",
            "input_type": "json",
            "input_format": "json schema",
            "output_type": "json",
            "processing_queue": "dns_queue",
            "is_automated": true,
            "autotrigger": true,
            "on_completion": [10]
            },
            {"id": 9,
            "name": "Blacklist malicious email addresses",
            "description": "...",
            "input_type": "json",
            "input_format": "json schema",
            "output_type": "json",
            "processing_queue": "email_queue",
            "is_automated": true,
            "autotrigger": true,
            "on_completion": [10]
            }]
         },
         {"name": "Revision Phase",
         "description": "...",
         "tasks": [{"id": 10,
            "name": "Prepare case summary",
            "description": "...",
            "input_type": "json",
            "output_type": "json",
            "is_automated": true,
            "autotrigger": true,
            "processing_queue": "ti_update_queue",
            "on_completion": [11]
            },
            {"id": 11,
            "name": "Create MISP event",
            "description": "...",
            "input_type": "json",
            "output_type": "url",
            "processing_queue": "misp_queue",
            "is_automated": true
            }]
      }]
}
```


*Obrázek 8. Šablona pracovního postupu pro proces Phishing*

Popis úkolů v šabloně pracovního postupu a samostatná šablona úkolu jsou velmi podobné, jediným rozdílem je, že v popisu úkolu v šabloně pracovního postupu je možné definovat i vztahy s ostatními úkoly prostřednictvím položek *on_completion*, *on_true*, *on_false* a *blocked_by*. Tyto vztahy v šabloně úkolu není možné definovat, jelikož šablona úkolu popisuje pouze jeden úkol a nemůže se na ostatní úkoly odkázat.

Popis úkolu obsahuje následující položky.

1. *id*: arbitrární celočíselný identifikátor platný pouze v rámci šablony. Identifikátor se používá pro definici vztahů mezi úkoly. Položka *id* je platná (a povinná) pouze v šabloně pracovního postupu, šablona úkolu jej nepoužívá.
2. *name:* jméno úkolu, povinná položka.
3. *description*: popis úkolu, preferovanou variantou je text ve formátu Markdown, který dokáže „Uživatelské rozhraní“ správně vykreslit, povinná položka.
4. *input_type*: typ vstupu.
5. *input_format*: dodatečná možnost validace vstupu, v obrázku šablony jsou JSON schémata vynechána.
6. *input_description*: popis vstupu.
7. *output_type*: typ výstupu.
8. *output_format*: dodatečná možnost validace výstupu.
9. *output_description*: popis výstupu.
10. *is_decision*: flag přepínající úkol do rozhodovacího režimu.
11. *is_automated*: flag přepínající úkol do režimu automatizovaného úkolu.
12. *processing_queue*: název fronty externího systému (Redis), do které se automatizovaný úkol serializuje.
13. *autotrigger*: flag přepínající automatizovaný úkol do režimu automatického spuštění po otevření úkolu.
14. *aggregates_results*: flag přepínající úkol do režimu agregátor.
15. *on_completion*: seznam identifikátorů úkolů, které se otevřou po (libovolném) dokončení daného úkolu.
16. *on_true*: seznam identifikátorů úkolů, které se otevřou v případě, kdy úkol má nastavený flag *is_decision* a úkol bude uzavřený s rozhodnutím TRUE.
17. *on_false*: seznam identifikátorů úkolů, které se otevřou v případě, kdy úkol má nastavený flag *is_decision* a úkol bude uzavřený s rozhodnutím FALSE.
18. *blocked_by*: platí pro režim agregátoru a označuje seznam identifikátorů úkolů, které nejsou rodiči tohoto úkolu, ale úkol blokují, tj. musí být vyřešeny před uzavřením tohoto úkolu. Výstup agregátoru musí být nastaven voláním speciální mutace, která ze všech zavřených úkolů (pokud mají blokační úkoly nastavený flag *is_decision* tak těch, které byly zavřeny s rozhodnutím TRUE) agreguje jejich výstupy.

Je nutné poukázat na to, že šablona udává určitou základní pevnou strukturu pracovního postupu, konkrétní pracovní postup je nicméně možné dále rozšiřovat standardním způsobem. Je tedy možné i do takto vytvořeného pracovního postupu přidávat další úkoly nebo fáze.

## 2.5. Komponenta pro konfiguraci poskytovatele identit Keycloak

**Cesta v adresáři výsledku:** `<backend-python/applications/csirtmu-auth-keycloak-fixture-loader>`

Mikro-služba inicializace dat pro Keycloak je jednoduchá pomocná služba, která je v rámci prototypu využita k jednoduché konfiguraci nástroje Keycloak, který slouží jako poskytovatel identity. Služba zajišťuje vytvoření správného prostředí. Připravuje realm, klienta, uživatele a nastavení je připraveno pro napojení na front-end.

### Závislosti

- Služby:
  - Poskytovatel identit Keycloak.
- Knihovny:
  - Knihovna pro konfiguraci poskytovatele identit Keycloak.

### Implementace

Žádoucí výslednou konfiguraci nástroje Keycloak je možné předepsat pomocí konfiguračního souboru ve formátu YAML. Struktura konfiguračního souboru je definována touto mikro-službou. Pro samotné nahrávání konfigurace do nástroje Keycloak byla vytvořena knihovna, která implementuje vlastního klienta pro Keycloak. Klient nabízí konkrétní dílčí metody, které spravují jednotlivé dílčí části konfigurace.

## 2.6. Uživatelské rozhraní ve frameworku Angular

### Klíčové koncepty

#### Odložené načítání modulů (lazy loading)

Jednou z nevýhod single-page aplikací (SPA) je potenciálně vysoký objem aplikace, protože ke kódu samotné aplikace musí být přibalen i framework, ve kterém je aplikace vyvinuta. Tento balík je načten jen jednou, při první navigaci do aplikace, nicméně toto úvodní načítání může být při komplexnějších aplikacích neúnosně dlouhé. Technikou částečně eliminující tuto nevýhodu je odložené načítání modulů. Při prvním načtení si prohlížeč ze serveru vyžádá pouze kód nezbytný k přístupu na požadovanou část aplikace. Tato část kódu typicky obsahuje kód frameworku, globální služby a komponenty či knihovny, které využívá zobrazená část aplikace. Při navigaci aplikací jsou postupně načítány potřebné moduly.

#### Dynamická konfigurace

Ve výchozím nastavení podporují Angular aplikace pouze statickou konfiguraci. Pro nasazení v jiném prostředí, nebo po každé úpravě konfigurace je nutné front-endovou aplikaci znovu sestavit. Sestavení aplikace zajistí dosazení konfiguračních hodnot na místa kódu, kde je konfigurace odkazována. Tento přístup má řadu nevýhod, jednou z nejvýznamnějších je jeho nekompatibilita s přístupem vytváření Docker obrazů.

Webová aplikace „Uživatelského rozhraní“ využívá alternativní způsob konfigurace. Tento způsob pošle před startem samotné Angular aplikace a vytvoření hlavní modulu HTTP dotaz na server, který vrátí konfigurační soubor ve formátu JSON. Aplikace tento soubor převede na typované rozhraní, které poskytuje všem komponentám a službám aplikace.

Použití dynamické konfigurace umožňuje vytvořit Docker obraz pouze jednou a poté ho opakovaně používat v různých prostředích se změněným konfiguračním souborem. V případě úprav konfigurace souboru není nutné na server nahrávat novou verzi aplikace, pouze upravený konfigurační soubor.

K implementaci dynamické konfigurace je využito utilit z knihovny Sentinel Common.

#### Chytré a prezentační komponenty

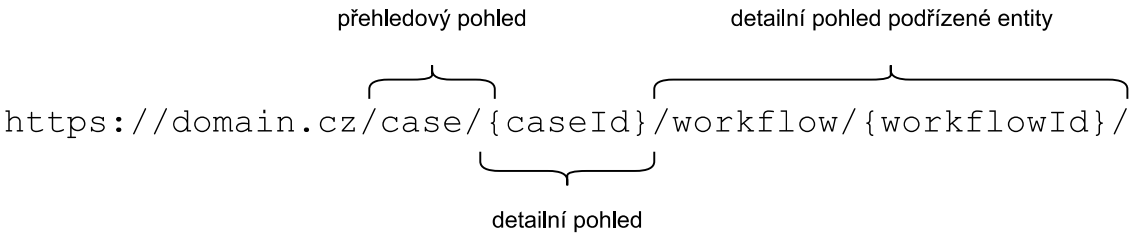
Rozdělení komponent na chytré a prezentační vychází z osvědčeného principu jedné odpovědnosti (single responsibility principle), který říká, že by každý objekt měl mít jednu konkrétní odpovědnost. V prostředí komponent uživatelského rozhraní se poměrně často stává, že jedna komponenta je zodpovědná za logiku získávání a zpracování dat a zároveň za prezentaci a prezentační logiku.

Principu chytrých a prezentačních komponent tuto logiku odděluje do dvou komponent. Chytrá komponenta je zodpovědná za získání asynchronních dat z datové služby, jejich zpracování a připravení pro prezentační komponentu. Prezentační komponenta na vstupu obdrží synchronní data a jejím jediným účelem je vykreslení dat, obsloužení událostí a zpracování prezentační logiky. Dodržováním tohoto principu se komponenty stávají přehlednější a testovatelnější.

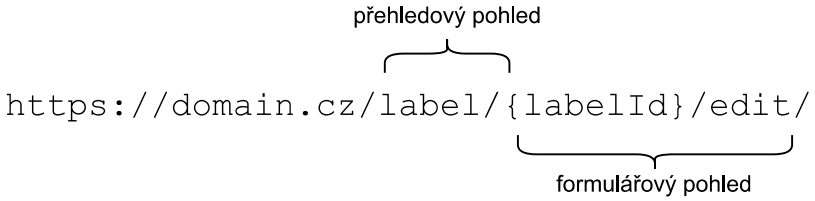
#### Navigace aplikací

Aplikace k navigaci využívá nativního routování frameworku Angular v kombinaci s odloženým načítáním modulů. Každá změna v adrese vyvolá načtení modulu a překreslení aktuálního pohledu. Jeden pohled vždy odpovídá jednomu modulu a jedné adrese.

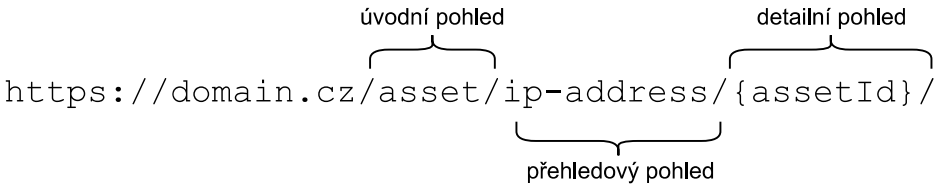
Ze struktury cesty lze odvodit agendu a typ pohledu. Vstupním pohledem do agendy je typicky pohled úvodní nebo pohled přehledový. Pokud adresa obsahuje specifikátor, jedná se o pohled detailní. Pokud má entita podřízené entity, mohou následovat další detailní pohledy. V případech, kdy je specifikátorem nahrazen slovem *create* nebo je specifikátor následován slovem *edit*, jedná se o formulářový pohled.




*Obrázek 9. Příklad URL navigující na detailní pohled podřízené entity v agendě správy případu*




*Obrázek 10. Příklad URL navigující na formulářový pohled v agendě štítků*




*Obrázek 11. Příklad URL navigující na detailní pohled v agendě aktiv*

### Ukládání stavu do URL

Kromě stavu navigace ukládá aplikace svůj stav do URL, přesněji do části URL nazývané dotaz (query params). Jako stav se ukládá například stránkování, filtrování, řazení, aktivní přepínací karta. Využití této vlastnosti je vhodné zejména z důvodů vetší uživatelské přívětivosti (ukládání do historie) a sdílení mezi uživateli. Při inicializaci aplikace z URL obsahující dotaz, si stránkovací, filtrovací, řadící a další služby načtou stav z URL a předají jej příslušným komponentám.

### Service Worker

Aplikace využívá service worker API, což je rozhraní podporované moderními webovými prohlížeči, které slouží jako proxy mezi webovou aplikací, prohlížečem a sítí. Mezi nejčastější využití API patří zachytávání síťové komunikace na úrovni prohlížeče, ukládání do mezi-paměti a push notifikace.

Angular poskytuje vlastní knihovnu a konfiguraci service worker, který při prvním zobrazení aplikace zajistí instalaci do prohlížeče a uložení aplikace do mezi-paměti. Všechna další načtení aplikace ze stejného prohlížeče jsou tedy rychlejší.

Výchozí service worker Angularu je rozšířen o vlastní implementaci, která odchytává požadavky prohlížeče na server a v případě, že se jedná o dotaz na obrázkové soubory uložené v zabezpečeném uložišti komponenty „Databáze případů“, přidá service worker k dotazu přístupový token uživatele. Toto řešení je nezbytné, protože prohlížeče posílají dotazy k zobrazení medií automaticky a není je tedy možné zachytávat v JavaScriptové aplikaci.

### GraphQL a Apollo Client

Aplikace „Uživatelského rozhraní“ využívá pro veškerou komunikaci se službami strany back-end rozhraní GraphQL. Konkrétně používá JavaScriptovou knihovnu Apollo Client a obalovací knihovnu Apollo Angular, která poskytuje aplikační rozhraní vhodné pro využití v Angular aplikacích.

Na základě analýzy často opakovaných pohledů v aplikaci a jejich vlastností (stránkování, filtrování, řazení) bylo vyvinuto několik generických tříd, které zjednodušují implementaci služeb posílajících GraphQL dotazy (queries), nebo mutace (mutations) a komponent konzumujících přijatá data. Tyto třídy jsou součástí mikro-frameworku ORION SCMFM.

Použití GraphQL, Apollo a vlastních rozšíření v mikro-frameworku ORION SCMFM v kombinaci s níže popsaným generováním typů výrazně zjednodušuje a zrychluje vývoj jednotlivých pohledů „Uživatelského rozhraní“. Zároveň toto řešení snižuje prostor k potenciálním chybám v době běhu aplikace zaviněných typovou nekompatibilitou mezi stranami front-end a back-end.

Apollo Link je součást knihovny Apollo, která umožňuje zachytávat a zpracovávat síťovou komunikaci, včetně úprav odesílaného dotazu. V aplikaci „Uživatelského rozhraní“ je Apollo Link využito k přidávání hlaviček pro autentizaci straně back-end.

### Generování typů

Pro GraphQL a Apollo existuje celá sada konfigurovatelných knihoven, které dokáží na základě GraphQL schématu, souborů fragmentů, dotazů a mutací vygenerovat nejen statické typy v jazyce TypeScript, ale i Angular služby implementující konkrétní dotazy a mutace v Apollo.

Veškerá síťová komunikace a mapování dat, což bývá často pracnou a repetitivní součástí kódu webových aplikací, je tedy v režii knihoven a generovaného kódu.

Ke generování kódu jsou použity knihovny GraphQL Code Generator, konkrétně ve verzi pro TypeScript a pluginy pro Apollo a Angular. Knihovna Code Generator umožňuje pokročilou konfiguraci a úpravu chování generátoru. Ve webové aplikaci „Uživatelského rozhraní“ je využita ke generování základních typů schématu, GraphQL dotazů, mutací a fragmentů, které jsou využívány jako rozhraní pro jednotlivé komponenty.

### Kolokace fragmentů

Kolokace fragmentů je způsob organizace kódu vhodný pro komponentové webové aplikace komunikující s GraphQL API. Použití technologie GraphQL a generování typů do jazyku TypeScript v kombinaci s přísným nastavením TypeScript kompilátoru je využito výhod statického typování.

Z důvodu minimalizace objemu dat přenášeného po síti je osvědčeným postupem žádat GraphQL API o nejmenší možnou množinu dat dostačující pro zobrazení požadované komponenty či stránky webové aplikace. Tento postup ale často v aplikacích s komplexní zanořenou strukturou komponent uživatelského rozhraní a využití znovupoužitelných komponent způsobuje, že vývojář, který píše GraphQL dotazy musí tuto nejmenší možnou množinu dat hledat. V případě změny v jedné z komponent se musí změna propagovat do všech dotazů, které tuto komponentu obsluhují. Tento proces zpomaluje vývoj a je náchylný k chybám.

Kolokace fragmentů využívá podobnosti mezi stromovou strukturou komponent uživatelského rozhraní a GraphQL dotazy, resp. fragmenty. Aplikace je tedy strukturovaná způsobem, kdy ve stejném adresáři, kde se nachází kód komponenty se nachází i příslušný GraphQL fragment. Pokud si aplikaci představíme jako strom komponent, můžeme říct, že pro komponenty, které jsou listy odpovídá nejmenší množina dat právě těm, která komponenta přímo používá. V případě komponent, které jsou uzly stromu, je nejmenší množinou dat právě ta, která přímo používá komponenta a všechny vnořené komponenty.

V případě použití generovaných typů a fragmentů je tedy dosaženo stavu, kdy fragment příslušící dané komponentě přesně odpovídá jejímu rozhraní. Tento fakt velmi zjednodušuje znuvupoužitelnost a testovatelnost komponent. Zároveň také ulehčuje refaktorování komponent.

## Příklad kolokace fragmentů

### Adresářová struktura

```
posts-page/
├─ posts-page.fragment.graphql
├─ posts-page.component.ts
├─ posts-page.operation.graphql
shared/
├─ user-icon-name/
│  ├─ user-icon-name.fragment.graphql
│  ├─ user-icon-name.component.ts
user-detail-page/
├─ user-detail-page-fragment.graphql
├─ user-detail-page.component.ts
├─ user-detail-page.operation.graphql
```

### Soubor `<user-icon-name.fragment.graphql>`

```
fragment userIconName on User {
	icon
	name
}
```

*Obrázek 12. Ukázka jednoduchého GraphQL fragmentu*

### Soubor `<user-detail-page.fragment.grapqhl>`

```
fragment userDetailPage on User {
	id
	address {
		street
		city
		country
	}
	...userIconName
}
```

*Obrázek 13. Ukázka složeného GraphQL fragmentu využívající jiný fragment pro sdílené atributy*

### Soubor `<posts-page.fragment.grapqhl>`

```
fragment postsPage on Post {
	id
	content
	user {
		...userIconName
	}
}
```

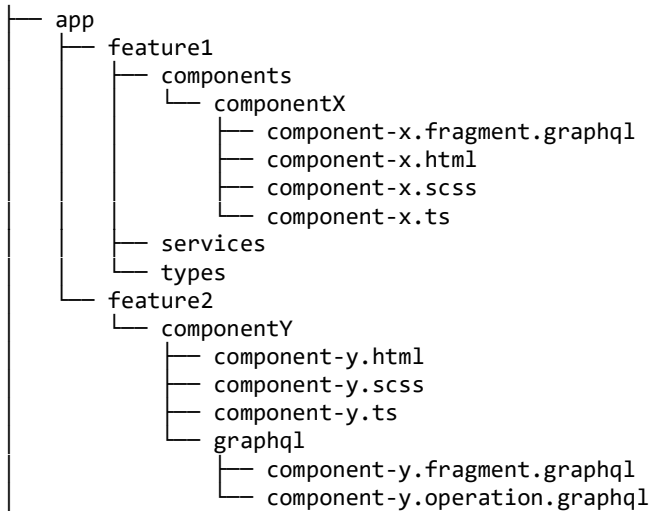
*Obrázek 14. Ukázka složeného GraphQL fragmentu využívající jiný fragment pro jeden z vnořených objektů*

### Testování

Při vývoji bylo využito manuálního testování a jednotkového testování, a to v aplikaci „Uživatelského rozhraní“ i mikro-frameworku ORION SCMFM. V obou případech jsou jednotkové testy součástí fáze nástroje průběžné integrace a průběžné dodávky. Jednotkové testy využívají knihovnu Jasmine a testovací prostředí Karma.

## Struktura

### Vzor adresářové struktury aplikace




## Možnosti rozšíření a úprav uživatelského rozhraní

### Změna konfigurace

Změnu hodnot konfigurace je možné provést v souboru *orion-config.json.* V případě změny struktury konfigurace je kromě změn v souboru *orion-config.json* nutné upravit i rozhraní `OrionConfig` v souboru *orion-config.ts*, na které se konfigurační soubor mapuje. K přístupu ke konfiguraci a jejím hodnotám slouží statická metoda `getConfig` třídy `OrionDynamicEnvironment`.

### Přidání komponenty

V případě, že se jedná o komponentu spadající pod existující agendu nebo její součást, je vhodné zkontrolovat adresářovou strukturu projektu a komponentu vytvořit v odpovídajícím adresáři. Pokud se jedná o komponentu nové agendy, je vhodné tuto agendu vytvořit podle výše zmíněného vzoru adresářové struktury. Pro umístění obecné, sdílené komponenty je vhodné využít adresář *shared,* nebo zvážit, zda není vhodné tuto komponentu integrovat do mikro-frameworku ORION SCMFM.

Pokud komponenta používá data odpovídající typům nebo částem typů z GraphQL schématu, měl by ke komponentě být přiložen GraphQL fragment a vygenerovaný model. Tento model pak komponenta využívá jako svůj datový vstup.

Dále je vhodné rozlišit, zda se jedná o chytrou komponentu, tedy komponentu komunikující s Apollo službami. V tomto případě je doporučeno dodržovat princip chytrých a prezentačních komponent. Chytrou komponentu je pak doporučeno rozšířit vhodnou generickou třídou z mikro-frameworku ORION SCMFM.

Z důvodu optimalizace výkonu používají všechny komponenty strategii detekce změn `ChangeDetectionStrategy.OnPush`.

### Postup pro přidání nové agendy

1. Vytvořit adresářovou strukturu dle výše zmíněných příkladů a vzorů. Do adresáře *components* jsou potom umisťovány všechny relevantní komponenty dle jejich logické hierarchie.
2. Vytvořit *index.ts* soubor či soubory v adresářích *components*, případně *services* a *types*.
3. Exportovat veřejné třídy v souborech *index.ts*.
4. V *tsconfig.json* přidat do sekce `paths` definici cest k jednotlivým souborům *index.ts* po vzoru existujících agend.
5. Přidání routovacích modulů pro novou agendu do adresáře *routes*.

6. Definice cesty a import hlavního routovacího modulu v `AuthSectionRoutingModule`.
7. Přidání instance nové agendy do navigačního menu v metodě `initAgendas` komponenty `AuthSectionComponent`.

## 2.7. Mikro-framework ORION SCMFM

**Cesta v adresáři výsledku:** `<frontend-angular/csirtmu-orion-microframework>`

### Sekundární vstupní body (secondary entry points)

JavaScriptové balíky mohou kromě primárního vstupního bod využívat také bodů sekundárních. Sekundární vstupní body jsou způsobem, jak zvýšit granularitu balíku a umožnit tak moderním kompilátorům odstranit z výsledné sestavené aplikace nepoužitý kód.

Příkladem primárního vstupního bodu je cesta `@orion/example-library`. Příkladem sekundárního vstupního bodu je cesta `@orion/example-library/my-component`. V případě importování z primárního vstupního bodu je do výsledné aplikace zahrnuta celá knihovna i přestože je v aplikaci využívána jen její malá část. Importováním ze sekundárních vstupních bodů, umožňuje vývojář kompilátoru optimalizovat velikost výsledné aplikaci a přibalit do ní jen importované části knihovny. Tato optimalizace se nazývá *tree shaking*.

Mikro-framework ORION SCMFM je kompilován a balen do npm balíku pomocí nástroje *ng-packagr* a je rozdělen do 31 sekundárních vstupních bodů.

### Stránkování, řazení a filtrování

#### Stránkování

Moderní webové aplikace velmi často používají techniku stránkování pro výpis polí dat. Jedná se o optimalizační techniku, kdy je již na úrovni databáze vybrána pouze požadovaná výseč záznamů a tím je zabráněno nadměrnému výpočetnímu a síťovému vytížení při operacích nad velkým objemem dat.

Stránkování se dále dělí na dva možné přístupy.

- Stránkování ofsetem (offset pagination),
- stránkování kurzorem (cursor pagination).

Stránkování ofsetem je implementačně jednodušší přístup, kdy klient požaduje určité množství záznamů s ofsetem, který udává, kolik záznamů v poli přeskočit. Tento přístup není vhodný pro dotazovaní se nad daty měnící se v reálném čase, protože může nastat situace, kdy jsou některé záznamy přeskočeny nebo zobrazeny dvakrát. Zároveň platí, že se zvyšujícím množstvím záznamů, klesá rychlost jejich načtení z databáze.

Stránkování kurzorem je přístup implementačně komplexnější, ale eliminující obě nevýhody ofsetového stránkování. Odlišnost spočívá v tom, že server kromě samotných dat vrací i kurzor (většinou ve formě hashe), což je unikátní identifikátor následujícího, či předešlého záznamu. Rychlost výběru záznamu z databáze je konstantní a nezávislá na objemu dat. Výsledky neobsahují žádné nepřesnosti, protože přístup nespoléhá na specifickou pozici v databázi, tak jak je tomu u předchozího přístupu.

ORION SCMFM poskytuje implementaci stránkování na straně klienta oběma způsoby. Mikro-framework zároveň využívá návrhový vzor strategie a přesnou implementaci volí na základě konfiguračního parametru, který je možné poskytnout na úrovni aplikace, nebo komponenty. Tento způsob implementace činí případný přechod mezi oběma implementačními přístupy velmi jednoduchým.

Stránkovací službu je možné použít ve dvou variantách. Jednou variantou je služba udržující stav stránkování v paměti aplikace. Dojde-li k obnovení stránky nebo navigaci na jinou stránku, stav je ztracen. Druhou variantou je stránkovací služba s ukládáním stavu do dotazové části URL. Stav stránkování je z URL obnoven i když dojde k znovunačtení stránky nebo otevření z URL.

Služba umožňuje měnit velikost stránky, navigovat na následující, nebo předchozí stránku, resetovat stránkování. Datový model umožňuje ke stavu stránkování přidat i stav řazení.

### Řazení

ORION SCMFM obsahuje stavové služby řazení. Podobně jako stránkovací služba má variantu ukládání v paměti, nebo v dotazu URL. Datový model řazení obsahuje atribut, podle kterého by měla data být seřazena a směr řazení.

### Filtrování

Služby filtrování jsou abstraktní generické služby, které udržují stav filtrování. Pro jejich použití je nutné abstraktní služby rozšířit a nahradit generický typ specifickým filtrem. Dále je nutné poskytnout implementaci určitých metod, dle zvolené varianty služby.

Varianty se podobně jako u služeb stránkování a řazení dělí na službu ukládající stav v paměti aplikace a službu ukládající stav v dotazu URL.

Ke službám filtrování mohou být připojeny služby filtrovacích žetonů (filter chips). Služba slouží ke konverzi stavu filtru na filtrovací žetony a úpravu stavu filtru při odstranění žetonu. Žetony jsou prvky uživatelského rozhraní reprezentující stav vybraného filtrování. Lze je využít například v kombinaci s komponentou panelu filtrů.

## Služby a komponenty integrované s Apollo

Knihovny Apollo Client a Apollo Angular sami o sobě velmi snižují množství kódu potřebného ke komunikaci s aplikačním rozhraním mikro-služeb back-end. Během prototypování „Uživatelského rozhraní“ bylo nalezeno několik opakujících se vzorů ve službách a komponentách komunikujících se službami Apollo, zejména v obsluze GraphQL dotazů.

Tyto opakující vzory byly implementovány formou abstraktních generických tříd služeb a komponent, které jsou navrženy ke vzájemné kooperaci a zapouzdřují opakující se kód a implementační detaily komunikace. Služby a komponenty můžeme rozdělit do následujících kategorií dle jejich charakteristik.

### Základní

*Základní službou je* `ApolloQueryService`*,* kterou je možné integrovat s komponentou rozšiřující třídu `EntitySmartOverviewDirective`. Téměř všechny ostatní služby a komponenty této sekce jsou postaveny na jejich základě.

Pro vlastní implementaci služby je nutné ji rozšířit a poskytnout datové typy `Query`, `QueryVariables` a *`Data`*, kde první typ popisuje GraphQL dotaz, druhý typ proměnné dotazu, typ `Data` poté popisuje typ, na který se výsledek dotazu mapuje (doporučeno je využít pro tento účel GraphQL fragment). Konkrétní typy pro `Query`, `QueryVariables` a `Data` je vhodné generovat z GraphQL dotazů a fragmentů pomocí nástrojů jako *GraphQL Code Generator.* Dále je nutné implementovat metody vytvářející proměnné dotazu a mapující výsledek dotazu typu `Query` na typ `Data`.

Služba v konstruktoru přijímá název operace, který je využit k vytváření chybových hlášek, a objekt možností, který umožňuje nastavit určité vlastnosti služby (zda se při chybě pokoušet opakovat dotaz, zda dotaz pravidelně opakovat a nastavení chování Apolla).

Služba poskytuje formou atributů pozorovatelného typu (observable) získaná data, stav načítání, chybový stav a stav znovu načítání.

Chytré komponenty využívající služby rozšiřující `ApolloQueryService` mohou rozšiřovat direktivu `EntitySmartOverviewDirective`. Tato direktiva provede inicializaci a obsluhu služby, namapuje pozorovatelné atributy na lokální, které lze využít v příslušné HTML šabloně a poskytne metodu pro zotavení po chybě.

### S podporou stránkování

Jedním z dalších často opakujících se vzorů jsou GraphQL dotazy s podporou stránkování. Mikro-framework obsahuje základní rozšiřitelnou službu podporující stránkování a službu transformující obecné stránkování na konkrétní rozhraní stránkování odpovídající aplikačnímu rozhraní generovanému „knihovnou DMT“ z Výsledku č. 1 [3], na kterém jsou postaveny mikro-služby strany back-end.

Oproti základní službě je navíc nutné poskytnout typ pro generický typ `OrderableBy`, tedy datový typ atributu, podle kterého bude výsledek filtrován. Typicky se jedná o typ řetězec, nebo výčet. V případě, že stránkování nepodporuje řazení, může být poskytnut typ `null`.

Rozšiřující služba musí implementovat metody převádějící stránkování a řazení na proměnné GraphQL dotazu a mapující výsledek dotazu na interní reprezentaci stránkovaných dat.

Chytré komponenty využívající stránkování mohou rozšířit direktivu `EntityPaginableSmartOverviewDirective`, která kombinuje stránkovací službu s výše zmíněnou službou pro GraphQL dotazy podporující stránkování. Direktiva je rozšířením výše zmíněné základní direktivy a poskytuje tedy stejné atributy, rozšířené o stav stránkování. Hlavní odpovědností direktivy je reakce na změny stavu stránkování a vyvolávání aktualizace proměnných GraphQL dotazu.

### S podporou filtrování

Dále ORION SCMFM poskytuje služby a direktivy podporující GraphQL dotazy s podporou filtrování, případně variant filtrování a stránkování, nebo filtrování a řazení. Tyto služby jsou kromě dalších funkcionalit primárně zodpovědné za konverzi klientského filtrování na filtrování podporované GraphQL aplikačním rozhraním.

Základní služby, případně služby poskytující stránkování, rozšiřuje generická abstraktní služba `ApolloFilterableService` (případně služby `ApolloFilterableOrderableService` a `ApolloPaginatedFilterableService`) o funkcionalitu konverze klientského filtru na filtr aplikačního rozhraní GraphQL. Pro použití služeb je potřeba poskytnout konkrétní typy na místo generických `ClientFilter` a `ApiFilter`. Poté je nutno implementovat metodu vytvářející tzv. konvertor, tedy instanci třídy převádějící klientský filtr na aplikační.

Příslušné direktivy pro chytré komponenty, tedy třídy `EntityFilterableSmartOverviewDirective`, `EntityFilterableOrderableSmartOverviewDirective` a `EntityPaginableFilterableSmartOverviewDirective`, slouží k propojení komponenty s příslušnou službou GraphQL a stránkovacími, řadícími, nebo filtrovacími službami. Direktivy všechny služby inicializují, poskytnou jejich rozhraní HTML šabloně a ošetřují určité implementační detaily, jako je resetování stavu stránkování po změně filtru či řazení.

### Tabulkové komponenty

Mikro-framework obsahuje i několik abstraktních generických direktiv, zapouzdřující funkcionalitu obsluhující prezentační komponentu tabulky z balíku Sentinel Components. Direktiva z dat na vstupu vytvoří instanci třídy tabulky a předá ji podřízené prezentační komponentě. Direktiva existuje ve variantách obyčejné tabulky, stránkované tabulky a tabulky podporující obyčejná i stránkovaná data.

### Přehledové prezentační komponenty

K direktivám pro chytré komponenty byly vytvořeny i direktivy ke komponentám prezentačním, které slouží k zapouzdření vstupních parametrů a výstupních událostní, které by měla komponenta využívat pro komunikaci s příslušnou nadřazenou komponentou.

## Popis komponent

Framework ORION SCMFM obsahuje sadu znovupoužitelných komponent, které byly definovány a vznikly zejména během vývoje „Uživatelského rozhraní“.

### Attribute

Komponenta Attribute je jednoduchou prezentační komponentou, která vypíše název atributu a na místo hodnoty atributu vloží projektovanou komponentu.

### Badge

Komponenta Badge je prezentační komponenta vykreslující prvek uživatelského rozhraní, který slouží jako odznak. Odznak slouží ke zvýraznění určité vlastnosti, většinou jedinečné, jiného prvku uživatelského rozhraní. V „Uživatelském rozhraní“ je komponenty využito k zvýraznění počtu vazeb na další entity.

### Button With Icon

Komponenta Button With Icon rozšiřuje komponentu Button sady komponent Angular Components. Vedle popisku zobrazuje konfigurovatelnou ikonu.

### Card

Komponenta Card zobrazuje projektovanou komponentu obalenou v elementu karta, která komponentu ohraničuje, volitelně zobrazuje titulek, popisek, hlavičku s obrázkem a uživatelské akce. Komponenta je integrována s komponentou uživatelských akcí ze sady komponent Sentinel Components.

### Color Block

Komponenta Color Block je jednoduchou prezentační komponentou, který na vstupu přijímá barvu a zobrazuje ji v ohraničeném prostoru.

### Date Form

Komponenta Date Form je vysoce konfigurovatelnou komponentou obsahující formulářový vstup pro zadávání data. Komponenta umožňuje datum zadat klasickým zápisem, nebo výběrem data z kalendářové komponenty. Vstupními parametry je možné konfigurovat vzhled komponenty, její validační vlastnosti, přepínat mezi módem pro zadávání pouze data, nebo data a času. Na výstupu komponenta vysílá událost při změně vybraného data, nebo změně validity data.

### Date Range Forms

Date Range Forms jsou dvě komponenty pro výběr rozsahu dat (od kdy do kdy). Z hlediska grafického zpracování komponenta zobrazuje dva vstupy pro výběr data (včetně kalendářové komponenty), jeden sloužící pro výběr dolní hranice rozsahu, druhý pro výběr horní hranice rozsahu.

Jedna z komponent vytváří instanci Angular třídy `FormGroup`, kterou následně používá k řízení a validaci formuláře. Tato komponenta je vhodná k využití v případech, kdy je komponenta samostatně stojícím prvkem. Druhá komponenta je komponentou podřízenou, která na vstupu přijímá instanci Angular třídy `FormGroup`. Tuto komponentu je vhodné použít v případě, kdy je potřeba ji zakomponovat jako součást rozsáhlejšího formuláře.

Obě komponenty na výstupu vysílají událost obsahující vybraný rozsah při změně každé změně vstupu.

### Due Date

Komponenta Due Date zobrazuje datum a uživatelsky přívětivou formou indikuje, zda již je dané datum v minulosti. V případě, že se jedná o datum v minulosti, je zbarveno varovně červenou barvou.

Ve stejném vstupním bodu se nachází i komponenta obalující komponentu Due Date do editovatelné obálky, vhodné pro použití v postranních kontextových panelech. Tato komponenta přepíná mezi zobrazením komponenty Due Date a níže zmíněné komponenty Editable Date.

### Editable

Komponenta Editable je obalovací komponentou, do které je možné projektovat komponenty či šablony ve variantě zobrazovací a editovací. Mezi těmito variantami je následně přepínáno pomocí akčních tlačítek. Při změně stavu komponenta vysílá událost a umožňuje tak nadřazené komponentě na tyto události reagovat.

### Editable Date

Komponenta Editable Date spojuje výše zmíněné komponenty Editable a Date Form do jedné, lehce použitelné komponenty. Komponenta v módu zobrazení vykreslí zvolené datum, ale pomocí akčních tlačítek umožnuje přepnout do editovacího módu, kde je možné datum upravit pomocí již známé komponenty Date Form.

### Entity Name

Komponenta Entity Name slouží k zobrazení názvu či jména entity. Entita se vyznačuje tím, že „Uživatelské rozhraní“ typicky obsahuje její detailní pohled. V tomto případě je možné komponentě předat prostřednictvím vstupních parametrů navigační data ve formátu pro Angular Router. Komponenta v takovém případě slouží k zobrazení názvu entity a zároveň jako navigační prvek.

Z prezentačního hlediska komponenta podporuje zkrácení názvu dle konfigurace maximálního možného počtu znaků. V případě, kdy název přesahuje zvolenou hranici, je název zkrácen a jeho zbytek nahrazen třemi tečkami. Plný název lze zobrazit v náhledu, který se zobrazí po najetí kursoru myši na komponentu.

### Enumeration Select

Enumeration Select je generická komponenta, která obaluje formulářovou komponentu rozbalovací nabídky z knihovny Angular Components. Integruje ji s datovými typy výčtového typu. Na vstupu přijímá pole hodnot libovolného typu rozšiřující typ řetězec a tyto hodnoty zobrazuje jako prvky rozbalovací nabídky. Po výběru z nabídky na výstupu vyšle událost s hodnotou zvoleného prvku.

### Filters

Vstupní bod Filters obsahuje několik komponent souvisejících s filtrováním. Jednou z komponent je panel filtrů, což je vysouvací panel s ovládacími prvky filtrování (akce filtrovat a zrušit filtrování) a prostorem pro zobrazení žetonů právě aktivních filtrů. Po rozbalení panelu se odhalí filtrovací formulář, který je do komponenty projektovaný.

Ze vstupního bodu je dostupná i samotná komponenta zobrazující seznam žetonů dle aktivních filtrů, která je využita v hlavičce komponenty filtrovacího panelu.

Poslední komponentou tohoto vstupního bodu je komponenta pro zadávání textového vstupu ve formuláři filtrů. Jedná se o komponentu zjednodušující použití komponenty pro textový vstup z knihovny Angular Components v kontextu implementace funkcionality filtrovacích formulářů.

### Inline List

Komponenta dle dat na vstupu vykreslí elementy v jednom řádku, oddělené konfigurovatelným separátorem. Jedná se o vhodný způsobem zobrazení textových seznamů. Zobrazení graficky pokročilejších prvků na místě jednotlivých elementů je možné dosáhnout poskytnutím vlastní HTML šablony, která je v komponentě projektována na místa jednotlivých elementů.

### Key-value List

Komponenta Key-value List zobrazuje seznamy dat typu pár klíč-hodnota. Klíčová část páru je odlišena použitím tučného neproporcionálního písma a podbarvením. Hodnotová část je zobrazena běžným písmem. Obě části jsou od sebe odděleny konfigurovatelným separátorem. Seznam je možno zobrazit ve sloupci, či řádku.

### Loading Skeleton

Loading Skeleton je vstupní bod knihovny obsahující řadu komponent, které je možné použít pro vytvoření tzv. skeleton loading screen. Tento pojem by se dal přeložit jako kostra načítané obrazovky. Jedná se o poměrně moderní prvek uživatelských rozhraní, jehož úkolem je zlepšení uživatelského komfortu při používání aplikace za účelem udržení pozornosti uživatele. Tento prvek by měl zobrazit pohled webové aplikace ve stavu načítání takovým způsobem, který uživateli co nejvíce a nejpřesněji nastíní, jak bude výsledný pohled vypadat. Toho je prakticky docíleno zobrazením v zásadě stejného rozložení, uspořádání a velikosti jednotlivých prvků uživatelského rozhraní daného pohledu s tím rozdílem, že samotná data, na jejichž načtení uživatel čeká (texty, obrázky, videa, ikony) jsou nahrazeny zástupnými prvky, které indikují stav načítání.

Tyto zástupné prvky v kostře pohledu vykazují vyšší uživatelskou přívětivost, jsou srozumitelnější a načítání dat je pocitově kratší, než je tomu v případě prázdné obrazovky, nebo rotujícího kolečka. Při konzistentním používání ve všech pohledech působí aplikace svižněji. Důležitou podmínkou použití načítacích koster je, že by kostra a všechny její zástupné prvky měly vizuálně co nejvíce připomínat výsledný stav po načtení.

Z tohoto důvodu obsahuje vstupní bod několik komponent, které byly navrženy dle předlohy existujících pohledů „Uživatelského rozhraní“. Komponenty jsou vysoce konfigurovatelné a odpovídají několika často používaným sdíleným komponentám, jako je karta, tabulka, postranní panel, přepínací karty, nadpis, text, či panel nástrojů.

### Nested Tree

Nested Tree je komponenta a řada tříd a služeb sloužící k zobrazení nekonečně vnořené stromové struktury. Komponenta podporuje vícero kořenů, uzly a listy stromu. Zobrazení jednotlivých uzlů je možno upravit poskytnutím vlastní HTML šablony, která je následně projektována na místo jednotlivých uzlů. Komponenta podporuje skrývání a odkrývání jednotlivých částí stromu, kdy prvky zobrazené po odkrytí podstromu jsou načítány asynchronně. V případě načítání podstromů, např. prostřednictvím HTTP komunikace, zobrazuje komponenta rotující kolečko a v případě chyby pak vypíše chybovou hlášku a umožní akci načtení opakovat. Pro případy stromů obsahující velké množství dat, umožňuje komponenta stránkovaní na všech úrovních stromu nezávisle na sobě.

Pro použití je potřeba rozšířit služby `OrionTreeDataService`, která slouží k načtení kořenů a uzlů stromu z datového zdroje, a `OrionTreeNodeManagerService`, která slouží k vytvoření instancí třídy reprezentující uzel.

### Panel

Panel je prezentační znovupoužitelná komponenta vysouvacího panelu. Panel se může nacházet ve sbaleném nebo rozbaleném stavu. Je vhodný pro zjednodušení a zpřehlednění komplexních pohledů „Uživatelského rozhraní“. Ve sbaleném stavu panel názvem a případně popiskem naznačuje co se nachází v jeho skryté části, ale nezabírá příliš mnoho prostoru daného pohledu. Po kliknutí se spustí animace, která panel rozbalí a zobrazí jeho obsah.

Na místo názvu, popisku a vnitřního skrytého obsahu je možné projektovat jakékoliv komponenty či HTML šablony. Panel podporuje zobrazení ovládacích prvků v pravé horní částí panelu, které jsou integrované s komponentou Sentinel Controls.

### Sidebar

Tento vstupní bod obsahuje několik komponent a direktiv pro zjednodušení vytváření kontextových postranních panelů. Hlavní komponentou je Sidebar Element, do které je možné projektovat dvě různé komponenty. Zobrazení jednotlivých komponent se pak přepíná podle toho, zda je postranní panel v rozbaleném, nebo sbaleném stavu. Rozbalený stav je vhodný pro projektování komponent v jejich plné verzi, naopak sbalený stav pro projektování komponent zobrazující daný prvek v omezené velikosti.

Typickou implementací je zobrazení atributu a jeho hodnoty v plné verzi komponenty a např. jen ikony reprezentující daný atribut a zkratky reprezentující jeho hodnotu ve sbaleném stavu.

### Tabs

Tabs je dalším poměrně rozsáhlým vstupním bodem mikro-frameworku. Sdružuje komponenty, služby a direktivy implementující funkcionalitu přepínacích karet.

Vstupní bod obsahuje službu, která slouží k definici identifikátorů jednotlivých přehledových karet a udržování informací o tom, která přepínací karta je aktivní. Služba je dostupná ve dvou variantách, první varianta ukládá stav do paměti aplikace a navigací na jinou stránku či obnovením stránky v prohlížeči je stav ztracen. Druhá varianta stav ukládá do části dotazu URL a umí stav z URL obnovit i v případě znovunačtení stránky nebo navigace na jinou webovou stránku a následné vrácení se. Při obnovení služba identifikuje aktivní přepínací kartu a její obsah je zobrazen.

Samotná komponenta pak vykresluje jednotlivé přepínací karty a jejich štítky dle poskytnutých HTML šablon. Každé šabloně je přiřazen unikátní identifikátor, který se musí shodovat s identifikátory poskytnutými přepínací službě. Komponenta tyto identifikátory spáruje s příslušnými HTML šablonami, které zobrazuje v případě, že je vybraná přepínací karta aktivní. HTML šablony využívají princip odloženého načítání a prohlížeč tedy vždy vykresluje jen HTML šablonu právě aktivní přepínací karty. Tato optimalizace umožňuje bez velké výpočetní zátěže přepínat mezi stovkami či tisíci přepínacích karet.

Aktivní přepínací karta je změněna kliknutím na příslušný štítek přepínací karty. Jednotlivé štítky přepínacích karet jsou umístěny v horizontálním pásu nacházejícím se nad obsahem aktivní přepínací karty. K vykreslení štítku přepínací karty je možné poskytnout vlastní HTML šablony, nebo využít komponenty Tab Label, která dle vstupních parametrů zobrazí ikonu, název a odznak (komponenta Badge).

### Text Match

Text Match je jednoduchou komponentou, která na vstupu přijímá řetězec reprezentující text a řetězec reprezentující vzor. Komponenta se pokusí najít v textu shodu dle vzoru (může se jednat o běžný řetězec nebo regulární výraz). Části textu, které odpovídají poskytnutému vzoru komponenta zvýrazní použitím silnějšího písma.

### Time Ago

Time Ago je jednoduchou obalující komponentou, která poskytnuté datum přemění na text pomocí roury z knihovny *ngx-timeago.* Komponenta tak přeloží specifické datum na text, říkající, kolik času od tohoto data uplynulo, nebo naopak, kolik času do daného data zbývá.

### Upload Files

Vstupní bod Upload Files obsahuje několik komponent sloužících jako prezentační komponenty pro implementaci funkcionality nahrání souboru či souborů. Je předpokládáno, že v konkrétní implementaci musí být komponenta doplněna chytrou komponentou, který na základě vysílaných událostí nahraje vybrané soubory do libovolného datového uložiště.

Komponenta podporuje několik konfigurací prezentačního charakteru, nastavení podporovaných akcí, nastavení výběru jen jednoho či několika souborů. Komponenta vysílá události v případech, kdy došlo ke změně vybraných souborů, provedení akce nahrání, či provedení akce zrušení.

Komponenta podporuje výběr souborů prostřednictvím systémového dialogového okna pro výběr souborů, nebo prostřednictvím akce táhni a pusť (drag and drop) do vyznačeného prostoru komponenty. Vybrané soubory lze odstranit. Nahrání souborů je zahájeno kliknutím na tlačítko *Upload*.

Nahrávání souborů je logicky rozděleno do komponent pro výběr souborů, zobrazení seznamu vybraných souborů, zobrazení akčních tlačítek. Vstupní bod obsahuje i hlavní komponentu, která pod-komponenty kombinuje v jednu. Pro pohodlnost a jednoduchost použití je možné použít jednotnou hlavní komponentu, v případech, kdy specifická implementace např. vyžaduje odlišné umístění prvků, či nějaké prvky vypouští, je možné použít jen některé z pod-komponent.

## Možnosti rozšíření

Následující sekce poskytuje několik návodů a možností, jak postupovat v případě, že by bylo potřebné mikro-framework SCMFM rozšířit.

### Přidání veřejné komponenty do existujícího vstupního bodu

Přidat novou komponentu do existujícího vstupního bodu je možné vytvořením a implementací komponenty. Dále je nezbytné komponentu definovat a exportovat v modulu. Dle charakteru komponenty a vstupního bodu je možné komponentu definovat a exportovat z již existujícího modulu, případně vytvořit modul nový. Zde se lze řídit pravidlem, že modul by měl být logickým sdružením podobné funkcionality.

Dalším nezbytným krokem je komponentu a případně nový modul exportovat ze souboru *public_api.ts,* což je speciální soubor, sloužící balíkovací knihovně jako indikátor veřejného rozhraní knihovny.

Další kroky jsou spíše dobrou praxí.

- Vstupy a výstupy komponenty prakticky tvoří její aplikační rozhraní, mělo by tedy být jednoznačné, srozumitelné a kvalitně zdokumentované.
- Komponenta, její rozhraní a příklady použití by měly být popsány v příslušném *README.MD* souboru.
- Komponenta by měla používat strategii detekce změn *`ChangeDetectionStrategy.OnPush.`*
- V případě, že komponenta používá jakékoliv stylování, které by mělo podléhat zvolenému tématu z knihovny Angular Components (např. barva písma, barva pozadí), je potřeba přidat ke komponentě téma a mixin.

### Změna veřejné komponenty

Při změně veřejné komponenty je potřeba dbát na to, zda se jedná o změnu, která nějak ovlivní aplikace, které na mikro-frameworku závisí a komponentu používají (tzv. breaking change). Může se jednat především o změnu vstupních parametrů, či výstupních událostí, ale i o příliš viditelné změny v prezentaci komponenty. V takovém případě je vždy vhodné se zamyslet, zda je změna nutná a zda má nějakou přidanou hodnotu. Pokud ano, je vhodné rozhraní, nebo celou komponentu označit jako zastaralé (deprecated) a nové rozhraní, či komponentu poskytovat vedle zastaralé varianty. Po určitém čase je možné zastaralé rozhraní, či komponentu zcela odstranit.

### Přidání nového vstupního bodu

Pro přidání vstupního bodu je potřebné vytvořit adresářovou strukturu dle následujícího vzoru.

```
├── csirtmu-orion-microframework
│   └── new-entry-point
│       ├── ng-package.json
│       ├── README.MD
│       └── src
│           ├── new-feature.component.html
│           ├── new-feature.component.scss
│           ├── new-feature.component.ts
│           ├── new-feature.module.ts
│           └── public_api.ts
```

Obsah soubor *ng-package.json* může být validní prázdný JSON soubor, tedy `{}`. Dalším krokem je vložení cesty k novému vstupnímu bodu do objektu `paths` v souboru *tsconfig.lib.json.*

```
"@csirtmu-orion/microframework/new-feature": ["./new-entry-point/src/public_api.ts"]
```

*Obrázek 15. Ukázka cesty k definici cesty nového vstupního bodu*

### Přidání závislosti na knihovně

Pokud je při nějaké úprav mikro-frameworku nutné přidat novou závislost na nějaké knihovně, je nezbytné tuto skutečnost uvést do metadat o npm balíku. Závislost na knihovnách jsou uváděny do souboru *csirtmu-orion-microframework/package.json* do sekce `optionalDependencies`.

### Přidání tématu komponenty

Pokud komponenta využívá stylování, které by mělo podléhat tématu (změna barvy textu, barvy pozadí), je vhodné ke komponentě vytvořit vlastní mixin téma komponenty. Použitím tématu komponenty je možné zobecnit komponentu tak, aby přejala určité stylování z hlavního tématu Angular Component, které je použité ve finální aplikaci. Příkladem může být situace, kdy je potřeba komponentě nastavit barvu pozadí. V případě nastavení konkrétní barvy, např. bílé, by komponenta nefungovala správně v aplikaci, která podporuje tmavý režim.

Soubor tématu komponenty by měl být vytvořen v adresáři *theming* v adresáři vstupního bodu. Poté je nutné přidat cestu k tomuto tématu sekce `exports` v souboru *csirtmu-orion-microframework/package.json.*

```
"exports": {
  "./new-component": {
    "sass": "./new-component/theming/_new-component-theme.scss"
  },
```

*Obrázek 16. Ukázka přidání cesty tématu balíkovacímu nástroji*

# 3. Instalační manuál

## 3.1. Úvod

### *Upozornění*

*I přesto, že jednotlivé výsledky a komponenty projektu lze užívat do určité míry samostatně, instalační manuál popisuje jejich instalaci (nasazení) do propojeného funkčního celku. Je to právě jejich společný provoz, jež nejlépe demonstruje jejich přidanou hodnotu. Následující text je tedy totožný pro všechny vytvořené výsledky typu „R – Software" – R1, R2, R3. Před započetím procesu nasazení a instalace proto doporučujeme jednotlivé instalační archivy výsledků sloučit do jedné kořenové složky.*

### Kubernetes

Z důvodu vysokého počtu a komplexity některých komponent vytvořeného projektu byl pro jeho nasazení zvolen přístup založený na technologii *Kubernetes***.** Tento přístup vyžaduje splnění několika podmínek službami, přičemž mezi jednu z nejzásadnějších patří jejich „kontejnerizace". Po kontejnerizaci služby vzniká tzv. obraz služby, jenž může nástroj Kubernetes stáhnout a spustit. Podmínkou je jeho nahrání do připraveného registru. Proces instalace je zajišťován pomocí nástroje *Helm***.** V tomto případě je však vhodné hovořit spíše o procesu nasazení.

Kubernetes (zkráceně K8s) je systém s otevřeným zdrojovým kódem, představený společností Google v roce 2014. Tento systém je určený k automatickému nasazování, škálování a správě skupin kontejnerů tvořících určitou aplikaci a stal se de-facto standardem v této oblasti.

Kubernetes obsahuje množinu entit, tzv. Kubernetes objektů, které reprezentují celkový stav klastru. Typicky se jedná o objekty popisující běžící kontejnerizované aplikace, zdroje dostupné těmto aplikacím a politiky jejich chování (například politika restartování a aktualizace).

Důležitou součástí Kubernetes je automatická aktualizace samotné aplikace nebo její konfigurace, ale i monitorování jejího stavu. Díky monitorování dokáže zamezit vzniku situace, kdy neexistuje ani jedna běžící instance aktualizované aplikace. Vývojáři aplikací mají možnost definovat vlastní funkce na kontrolu stavu aplikace, které pak Kubernetes k monitorování využívá.

Stálé úložiště je ke kontejnerům automaticky připojováno a pomocí rozšiřujících Kubernetes aplikací lze zabezpečit a spravovat replikaci úložného prostoru.

Pro ukládání a poskytování citlivých údajů aplikacím existuje speciální typ objektu, díky čemuž není potřeba tyto údaje vkládat do obrazu kontejneru. Použití těchto objektů snižuje riziko odhalení údajů použitých v nasazeném kontejneru i jeho obrazu.

Aplikace mají přiřazenou IPv4 (popř. IPv6) adresu a díky internímu DNS systému dokáže Kubernetes adresovat jednotlivé instance aplikací a rozdělovat mezi ně zátěž (tzv. load-balancing). Pro adresaci z externího prostředí je nabízen typ objektů chovajících se jako brána, a na který je možné nasměrovat skutečný DNS záznam a přiřadit mu TLS certifikát.

V katastrofickém scénáři selhání stroje dokáže K8s aplikaci nasadit znovu na jiný stroj (uzel) klastru, a to za předpokladu, že je daný stroj dostupný a splňuje určité podmínky. V Kubernetes lze definovat limity využití hardwarových zdrojů jednotlivými aplikacemi a horizontálně je škálovat.

Mezi konkrétní příklady Kubernetes objektů patří:

- **Pod –** nejmenší vytvořitelná výpočetní jednotka v Kubernetes. Tvoří ho skupina jednoho či více kontejnerů se sdíleným úložištěm a síťovými zdroji.
- **Deployment –** nástroj pro správu podů. Umožňuje deklarativně definovat požadovaný stav aplikace, který je automaticky propagovaný Deployment kontrolérem do klastru.
- **Service –** je abstrakce umožňující přistupování k množině podů jako k síťové službě.
- **Ingress –** směruje HTTP a HTTPS komunikaci mimo klastr do/z připojené služby.

- **Secret –** obsahuje citlivé údaje, které se nevkládají přímo do obrazu kontejneru. Vkládají se do kontejneru ad-hoc nebo automatizovaně během nasazování.

## Popis použitých Kubernetes instancí

Pro odladění instalace a nasazení výsledků byl vývoj prováděn na dvou odlišných systémech s různou konfigurací. Používané Kubernetes klastry měly následující konfiguraci.

### Fyzická instance

- **1x fyzický uzel**
  - Ubuntu 22.04.1 LTS
  - 754 GB RAM
  - Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz

### Virtualizovaná instance (OpenStack)

- **3x virtuální řídící uzel**
  - Ubuntu 20.04.4 LTS
  - 16 GB RAM
  - 4 VCPU
- **18x virtuální výpočetní uzly**
  - Ubuntu 20.04.4 LTS
  - 64 GB RAM
  - 16 VCPU

## 3.2. Helm

Helm je podpůrný nástroj pro automatické vytváření, balení, konfiguraci a nasazení aplikací nebo služeb do Kubernetes prostředí. V analogickém přirovnání Kubernetes k operačnímu systému počítače by Helm figuroval jako správce balíků (např. Apt pro Debian, Homebrew pro Mac).

Helm umožňuje vytváření izolovaných balíků, tzv. Helm chartů (předpisů), obsahujících všechny potřebné zdroje pro nasazení dané aplikace. Níže uvádíme ukázkovou strukturu Helm chartu.

```
Example-chart
├── files/
│   └──  something.conf.tpl
├── templates/
│   └──  app.yaml
├── Chart.yaml
└── values.yaml
```

Složka *templates* obsahuje šablony, které pomocí šablon jazyka Go a konfiguračních hodnot ze souboru *values.yaml* vytvoří definice Kubernetes objektů. Ty jsou pak použité pro nasazení jednotlivých komponent.

Soubor *Chart.yaml* obsahuje informace o daném chartu – jeho název, verzi, odkazy na jiné charty potřebné pro spuštění dané služby, či informace o autorech.

Složka *files* je využívaná k umístění dodatečných souborů obsahujících například předpřipravená data, kterými je aplikace inicializována, nebo konfigurační soubory aplikací.

Jelikož je proces nasazení ve většině služeb velmi podobný, vznikl tzv. *generic-helm-chart*, obsahující společnou logiku nasazování aplikací. *Generic-helm-chart* je používaný jako závislost v Helm chartech služeb, které jej k nasazení využívají. Takové charty je následně možné spojit pomocí dalšího nadřazeného chartu – *umbrella-chart*. Implementace takového chartu spočívá ve specifikaci jednotlivých chartů služeb v *Chart.yaml* jako závislostí a ve *values.yaml* obsahuje jejich konfiguraci použitou při nasazení. Takový proces umožňuje dekompozici složitého procesu nasazení do méně komplexních modulů, jež jsou následně integrované dohromady.

### Generický Helm chart

**Cesta v adresáři výsledku**: `<helm-charts/generic-helm-chart>`

Za účelem sjednocení struktury, znovupoužití kódu, definice proměnných a šablonování Kubernetes objektů byla vytvořena generická knihovna, používaná jako závislost v Helm chartech většiny bezstavových služeb projektu.

Mezi šablonované objekty, které je možné využít v nadřazených chartech a následně nasadit do klastru patří:

- Certificate
- ConfigMap
- Deployment
- Ingress
- ServiceMonitor
- Secret
- Service

Celkově se v nadřazeném chartu musí vykonat následující kroky, aby bylo možné knihovnu využít:

1. V *Chart.yaml* se musí knihovna specifikovat jako závislost.
2. Je nutné vytvořit soubor *app.yaml* v *templates* složce, obsahující libovolné potřebné šablony příkazem *include*, například . *{{ include "lib-chart.deployment" . }}*, jelikož Deployment šablona je definovaná příkazem *{{- define "lib-chart.deployment" -}}.*

### Nahrání do registru

1. Ve složce obsahující soubor *Chart.yaml* spustit příkaz pro vytvoření balíku s Helm chartem.

   ```
   $ helm package .
   ```

2. Nahrání balíku do registru.

   ```
   $ curl --request POST --user <jméno>:<heslo> \
   --form "chart=@generic-helm-chart-<verze>.tgz" "<adresa_helm_registru>"
   ```

## 3.3. Předpoklady pro instalaci a nasazení

Proces nasazení platformy vyžaduje existenci registru, který je využíván k uložení a sdílení vyvinutých artefaktů. Konkrétně se jedná o kontejnerizované obrazy, Helm charty a Python balíky.

Během vývoje platformy byla pro tyto účely využívaná služba GitLab, která mimo jiné poskytuje i registry určené pro tyto účely. Další ze známých alternativ ke GitLab registrům je například Sonatype Nexus, podporující uložení všech dříve zmíněných artefaktů.

Dalšími z předpokladů úspěšného nasazení platformy a spuštění připravené ukázky je přístup ke Kubernetes klastru se všemi potřebnými závislostmi a k S3-kompatibilnímu úložišti.

### Instalace S3-kompatibilního úložiště

Jakožto S3-kompatibilní úložiště bylo zvoleno MinIO – objektové úložiště primárně používané na ukládání nestrukturovaných dat.

Postup instalace MinIO:

1. Přidání oficiálního Bitnami repositáře.

   ```
   $ helm repo add bitnami https://charts.bitnami.com/bitnami
   ```

2. Instalace.

   ```
   $ helm install s3 bitnami/minio
   ```

3. Dle vytvořené domény se v souborech *values.yaml* „základní infrastruktury“ a „orchestrační logiky“ upraví hodnoty:
   - ***Host –*** adresa MinIO Ingressu.
   - ***Port –*** port MinIO Ingressu.
   - ***AccessKey*** a ***SecretKey –*** hodnoty z proměnných *auth.rootUser* a *auth.rootPassword.*
   - ***BucketName –*** jméno složky kam budou data ukládaná.

### GitLab

Služba GitLab funguje, kromě poskytování registrů pro ukládání artefaktů, především jako komplexní platforma pro kolaborativní vývoj softwaru. Je založená na technologii Git a poskytuje řadu podpůrných služeb, konkrétně třeba již dříve zmiňované registry pro artefakty, prostor pro poskytování statických stránek nebo službu GitLab CI (Continuous Integration).

GitLab CI je nástroj umožňující kontinuální spouštění integračního procesu pro dané projekty. Při vývoji byl tento nástroj využíván na zprostředkování automatického spouštění testů, správu verzí a automatické generování artefaktů v podobě různých druhů balíků, obrazů kontejnerů a dalších.

### Použité nástroje

Během vývoje byly použity níže uvedené nástroje, u nichž je garantovaná funkčnost jednotlivých služeb. Je velmi pravděpodobné, že instalační procesy a aplikace budou funkční i s jinými než uvedenými verzemi.

- **Kubectl –** v1.26.0
- **Helm –** 3.10.3
- **Kubernetes –** v1.24.4+rke2r1
- **Docker –** 20.10.22
- **Java –** 17.0.5
- **Apache Maven –** 3.6.3

## 3.4. Příprava artefaktů

Před samotnou instalaci je potřebné jednotlivé artefakty manuálně připravit a nahrát je do dedikovaného registru odkud budou přístupné K8s klastru. V několika následujících podsekcích je popsaný proces přípravy jednotlivých artefaktu na základě jejich typu.

Přehled jednotlivých služeb a potřebných artefaktů je uvedený v příslušné sekci (3.9). Přehled konkrétních příkazů pro registr v platformě GitLab je také uveden v příslušné sekci (3.11).

### Python knihovny

Je třeba zvolit registr, který podporuje ukládání Python balíků, například PyPi. Pro nahrání balíku je třeba vykonat následující kroky:

1. Nastavení cílového registru, do kterého se balík nahraje.

   ```
   $ poetry config repositories.ORION <adresa_registru>/pypi
   ```

2. Získání přihlašovacích údajů k registru a jejich uložení do nástroje Poetry.

   ```
   $ poetry config http-basic.ORION <jméno> <heslo>
   ```

3. Vytvoření balíku.

   ```
   $ poetry build -f wheel
   ```

4. Nahrání balíku do registru.

   ```
   $ poetry publish -r ORION
   ```

Tento případ se týká Python knihoven nacházejících se v podskupině *backend-python/libraries*.

### Python aplikace

V případě Python aplikací (služeb) využívajících takové knihovny je potřeba vykonat následující:

1. Nastavení zdrojového registru, odkud se stáhnou Python knihovny.

   ```
   $ poetry source add ORION <adresa_registru>/pypi
   ```

2. Případně nastavení přihlašovacích údajů pro registry.

   ```
   $ poetry config http-basic.ORION <jméno> <heslo>
   ```

3. Instalace samotné knihovny (v tomto případě se instaluje knihovna <jméno_balíku> z registru ORION).

   ```
   $ poetry add --source ORION <jméno_balíku>
   ```

Následně se musí vytvořit a nahrát kontejnerový obraz Python aplikace do registru artefaktů.

### Apache Airflow pluginy

V rámci projektu orchestrační logiky se nachází specifikace závislých Apache Airflow pluginů ve složce *files/airflow.* V souboru *pyproject.toml* jsou pomocí nástroje Poetry definovány verze všech knihoven a *poetry.lock* obsahuje fixované verze všech tranzitivně závislých knihoven. Pro instalaci přídavných knihoven do Apache Airflow je používán následující přístup:

1. Aktualizace verzí, přidání nových knihoven do *files/airflow/pyproject.toml.*

   ```
   $ poetry add --source ORION <jméno_balíku>
   ```

2. Vygenerování nového *poetry.lock*.

   ```
   $ poetry update
   ```

3. Exportování závislostí do *files/airflow/requirements.txt* (je nutné spustit ve složce *files/airflow*).

```
$ poetry export -f requirements.txt --without-hashes --with-credentials \
--output requirements.txt
```

Pro nasazení Apache Airflow je používán *requirements.txt* soubor pro automatizovanou instalaci daných závislostí během startu kontejneru.

Alternativně je možné ve vygenerovaném *requirements.txt* změnit zdrojové registry balíků, a to přidáním dalšího řádku s *--extra-index-url <adresa registru>* za první řádek.

## Apache Spark (Java)

Specifickou aplikací je *spark-iceberg-writer,* který je vytvořený v jazyku Java a pro generovaní jeho kontejnerového obrazu využívá Maven plugin **Jib**. Pro instalaci je potřebné spustit následující příkaz:

```
$ mvn package jib:build -DskipTests=true \
-Djib.to.auth.username=<jméno> -Djib.to.auth.password=<heslo> \
-Djib.to.image=<adr_registru>/<cesta_projektu>/csirtmu-spark-iceberg-writer:<verze>
```

## Docker

Pro nahrání obrazů do repositáře je třeba vykonat následující kroky:

1. Přihlášení se do Docker registru.

```
$ docker login -u <jméno> -p <heslo> <registr>
```

2. Vytvořeni obrazu.

```
$ docker build \
--build-arg POETRY_HTTP_BASIC_ORION_USERNAME=token \
--build-arg POETRY_HTTP_BASIC_ORION_PASSWORD=<heslo> \
-t <registr>/<název_obrazu>:<verze> <cesta_k_docker_souboru>
```

3. Nahrání obrazu do registru.

```
$ docker push <registry>/<název_obrazu>:<verze>
```

V případě služby *csirtmu-similarity-milvus* je potřeba nejprve spustit příkaz, který postaví tzv. *base* obraz.

```
$ docker build -t csirtmu-similarity-milvus-base \
--build-arg POETRY_HTTP_BASIC_ORION_USERNAME=token \
--build-arg POETRY_HTTP_BASIC_ORION_PASSWORD=<heslo> \
-f Dockerfile.base .
```

## Helm

V případě Helm chartů je třeba postupovat následovně:

1. Před instalací prvního chartu vykonejte následující příkaz.

```
$ helm repo add --username <jméno> --password <heslo> gitlab-orion
<adresa_helm_registru>
```

2. V případě, že je služba založená na *Generic Helm Chartu,* je nutné v souboru Chart.yaml změnit verzi *generic-helm-chart* na verzi, která se nachází v použitém registru.

3. Ve složce obsahující soubor *Chart.yaml* spustit příkaz pro vytvoření balíku s Helm chartem.

```
$ helm package –-dependency-update .
```

4. Nahrání balíku do registru.

```
$ curl --request POST --user <jméno>:<heslo> \
--form "chart=@<jméno_projektu_s_verzi>.tgz" "<adresa_helm_registru>"
```

## 3.5. Příprava K8s klastru

Před samotnou instalací výsledků projektu je třeba inicializovat K8s klastr pomocí dodatečných komponent rozšiřujících možnosti základní instalace klastru. Jedná se o sadu námi zvolených nástrojů, doplňujících možnosti použití K8s klastru pro vývoj libovolné služby.

### Komponenty

- **cert-manager –** kontrolér certifikátů, klíčů a certifikačních autorit pro Kubernetes, který dokáže vydávat certifikáty podepsané veřejnou nebo privátní certifikační autoritou (CA), reprezentovanou v Kubernetes jakožto objekt typu ClusterIssuer.
- **ExternalDNS –** aplikace využívající standardu RFC2136 pro dynamickou aktualizaci DNS záznamů směřujících do klastru.
- **MetalLB –** poskytuje implementaci síťového distributoru zátěže (Load Balancer) pro klastry tvořené fyzickými stroji. Dovoluje tedy vytvářet vlastní Service typu LoadBalancer mimo poskytovatele cloudových služeb.
- **Longhorn –** aplikace poskytující distribuované blokové úložiště pro Kubernetes s vysokou dostupností a podporou inkrementálních záloh.
- **Ingress NGINX Controller –** Ingress kontrolér, který poskytuje Ingress třídu. Bez Ingress kontroléru není možné vytvářet objekty typu Ingress.
- **Zalando Postgres operátor –** zabezpečuje instalaci a správu PostgreSQL databází.
- **Spark operátor –** vytváří a spravuje instance Apache Spark dávkových a proudových úloh.
- **Strimzi operátor –** vytváří a spravuje Apache Kafka klastry spolu s možností deklarativní definice příslušných Kafka témat.

### Operátory

Kubernetes operátor je návrhový vzor umožňující správu komplexnějších aplikací. Obecně řečeno, samotný operátor je kontrolér rozšiřující Kubernetes API, který umožňuje správu instancí komplexnějších aplikací během celého jejich životního cyklu a napříč celým klastrem. S operátorem je spojen seznam různých konfiguračních možností ve formě vlastního Kubernetes objektu, na které operátor reaguje.

### Storage Class

Storage Class je způsob, kterým se popisují třídy poskytovaných úložišť. Různé třídy mohou poskytovat např. odlišné politiky zálohovaní.

Každá třída obsahuje atributy *provisioner, parameters* a *reclaimPolicy*, které se použijí při dynamické alokaci persistentního úložiště.

- **Provisioner –** který plugin se má použít pro alokaci úložiště.
- **Parameters –** parametry pluginu, např. počet replik, typ souborového systému apod.
- **ReclaimPolicy –** politika, která určuje způsob zacházení s uvolněným, dříve alokovaným úložistěm.

V kontextu projektu byl použitý Longhorn jako poskytovatel distribuovaného blokového úložiště. StorageClass pro Longhorn kompatibilní s naší instalací je uvedená v sekci 3.10.

Výše zmíněnou definici objektu je potřeba do klastru nahrát následujícím příkazem.

```
$ kubectl apply –f storageclass.yaml
```

Následně se jméno třídy, podle přílohy v sekci 3.10, musí specifikovat v souboru *values.yaml* v proměnné *storageClass (*viz kapitola o *values.yaml*).

## Cluster Issuer

Cluster Issuer je objekt poskytovaný *cert-managerem* obsahující odkaz na certifikační autoritu, pomocí níž jsou podepisovány certifikáty, které jsou následně využívaný Ingress objektem k šifrování provozu. Aplikace, které jsou součástí projektu, očekávají, že v klastru bude dostupný alespoň jeden Cluster Issuer. V případu, že se objekt jmenuje jinak než *snakeoil-salesman* je nutné změnit jeho název ve *values.yaml* u každého z Helm chartů co jej používají (viz kapitola o *values.yaml*). Do klastru jej lze přidat pomocí následujícího příkazu (obsah souboru *clusterissuer.yaml* je v příkladu v sekci 3.10).

```
$ kubectl apply –f clusterissuer.yaml
```

## 3.6. Instalace

Instalace výsledků je rozdělena do tří částí. Jak již bylo uvedeno, toto rozdělení neodpovídá jednotlivým výsledkům a bylo zvoleno z důvodu usnadnění vývojového a instalačního procesu projektu jako celku.

### Příprava dat

**Cesta v adresáři výsledku**: `<helm-charts/data-pipeline/cc-data>`

Společně se zdrojovými kódy je přibalena i složka obsahující ukázková data, kterou je potřeba nahrát do připraveného S3 úložiště. Jedná se o data zachycená během kyberbezpečnostního cvičení v roce 2019 [4]. Složka s těmito daty obsahuje síťové toky, Linuxové a Windows logy.

```
$ mc alias set myminio https://minioserver.example.net ACCESS_KEY SECRET_KEY
$ mc mb myminio/cc-data
$ mc mb myminio/orion-data
$ mc cp IPFlowEntry.json.gz myminio/cc-data/ipfix.json.gz
$ mc cp SyslogEntry.json.gz myminio/cc-data/linux.json.gz
$ mc cp WinlogEntry.json.gz myminio/cc-data/windows.json.gz
```

### Příprava souboru *values.yaml*

Jelikož prostředí, do nichž můžou být výsledky nasazené, můžou mít různé specifické nastavení, je třeba před spuštěním instalace upravit hodnoty souborů *values.yaml* jednotlivých Helm chartů na základě následujících doporučení.

Nejdůležitější je korektně nastavit hodnoty pod atributem *image*, *registry* což je adresa registru, ve kterém jsou uložené obrazy aplikací a název obrazu samotného, *image.* Verze obrazu udávaná atributem *tag* je shodná s verzí aplikace samotné a tagem v repositáři aplikace.

Mezi další často měněné atributy patří *ingress,* který udává adresu, pod kterou bude služba vystavena. Pod-atributy jsou *host*, *path* a *annotations,* které lze využít pro konfiguraci např. ExternalDNS a cert-manageru.

Všechny aplikace lze horizontálně škálovat pomocí proměnné *replicas*, kterou lze nastavit počet instancí aplikace, které ve klastru poběží, tyto instance jsou automaticky umístěny na různé uzly. V klastrech jen s jedním uzlem by tento atribut měl být nastaven na hodnotu *1*.

Konfigurace se službám poskytuje pomocí atributů *env* a *config*, většina služeb umožňuje přepisovat stejnou konfiguraci pomocí jak proměnných prostředí, tak pomocí konfiguračních souborů v jazyce YAML. Proměnné prostředí jsou specifikované pomocí seznamu objektů ve formátu *name:value*. Konfigurační soubory pod atributem *config* můžou mít jeden ze tří různých typů:

- **Value –** ve *value* je hodnota, která bude použita jako obsah konfiguračního souboru.
- **File –** *ve value* je cesta ke konfigurační souboru uvnitř Helm chartu.
- **ConfigMap –** ve *value* je název existující *ConfigMap.*

Ostatní atributy by měly mít vhodně nastavené výchozí hodnoty. Pro konkrétní automatickou přípravu TLS certifikátů je třeba mít nastavenou hodnotu *clusterIssuer* na hodnotu *snakeoil-salesman (*viz. kapitola o přípravě závislostí v klastru). Obdobně je třeba nastavit hodnotu s*torageClass* na *longhorn*.

## Instalace základní infrastruktury

**Cesta v adresáři výsledku**: `<helm-charts/data-platform>`

1. Přidání Helm repositáře, ve kterém se nachází závislosti (Helm charty).

   ```
   $ helm repo add --username <jméno> --password <heslo> gitlab-orion \
   <registr>
   ```

2. Stažení závislostí.

   ```
   $ helm dependency update
   ```

3. V souboru *creds.yaml* je nutné nahradit všechny zástupní hodnoty *SHOULD_BE_CHANGED za* skutečné přístupové údaje.
4. V souboru *values.yaml* v sekci *Trino.ingress.hosts* a *Superset.ingress.hosts* je v případě potřeby možné změnit názvy jejich domén.
5. Nainstalování základní infrastruktury může trvat několik minut a může selhat v případě, že výpočetní síla klastru je nedostatečná. Z tohoto důvodu je v instalačním příkazu nastavený parametr *timeout* na 15 minut.

   ```
   $ helm upgrade --install --timeout 15m0s -f values.yaml -f creds.yaml \
   --create-namespace -n orion-infrastructure orion .
   ```

6. Aby mohl Apache Superset komunikovat s Trinem, je nutné Superset manuálně nakonfigurovat (*Data -> Databases -> iceberg (edit)*):
   a. Povolit *user impersonification (Advanced -> Security -> Impersonate logged in user)*
   b. Nastavení *Security tab -> SECURE EXTRA* musí vypadat následovně:

```
{
    "auth_method": "basic",
    "auth_params": {
        "username": "admin",
        "password": "CHANGE_ME_123"
    }
}
```

## Spuštění transformace nahraných dat

**Cesta v adresáři výsledku**: `<helm-charts/data-pipeline>`

Druhým krokem je spuštění procesu transformace nahraných dat a jejich ukládání do tabulek. Pro tyto účely existuje uvedený Helm Chart, který obsahuje definice příslušných Spark aplikací.

1. Přechozí kroky musí být úspěšně splněné.
2. Nahradit všechny zástupní hodnoty *SHOULD_BE_CHANGED* v souboru *creds.yaml* skutečnými hodnotami.
3. Podle potřeby nastavit hodnotu atributu *streaming* na *True* nebo *False*.
4. Spustit instalační proces, který nainstaluje vše potřebné do jmenného prostoru *orion-pipeline.*

   ```
   $ helm upgrade --install -f values.yaml -f creds.yaml --create-namespace -n
   orion-pipeline orion .
   ```

## Instalace mikro-služeb a orchestrační logiky

**Cesta v adresáři výsledku**: `<helm-charts/orchestration-demo>`

Proces instalace třetí, tedy poslední části, je téměř totožný s první částí. Rozdělené jsou hlavně z důvodu snížení komplexnosti nasazování během vývoje.

1. Přidání Helm repositáře, v němž se nachází potřebné závislosti (Helm charty).

   ```
   $ helm repo add --username <jméno> --password <heslo> gitlab-orion <registr>
   ```

2. Přidání oficiálních Helm repositářů externích služeb.

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm repo add bootc https://charts.boo.tc
$ helm repo add mattermost https://helm.mattermost.com
```

3. Stažení závislostí.

```
$ helm dependency update
```

4. V souboru *internals.yaml* se nastaví:
    a. Hodnota proměnné *secrets.password* na heslo k registru s obrazy kontejnerů.
    b. V souboru *templates/configmaps/dns-firewall.yaml* v sekci *named.conf* rozsah adres, které budou mít práva vykonávat operace na serverech.

```
acl "allowed-clients" {localhost; localnets; <ipv4-rozsah>/<ipv4-maska>;
<ipv6-rozsah>/<ipv6-maska>;};
```

    c. Doménová jména Ingresů dle potřeby.
5. V souboru *externals.yaml* se nastaví:
    a. Hodnota proměnné *PACKAGE_REGISTRY_TOKEN* služby Apache Airflow.
    b. Doménová jména Ingresů dle potřeby.
6. Spuštění instalačního příkazu.

```
$ helm upgrade --install --timeout 15m0s -f internals.yaml -f
externals.yaml --create-namespace --namespace orion-orchestration orion .
```

7. Proces instalace závislostí závisí na výkonu klastru, který lze většinou považovat za dokončený, jakmile je aplikace Airflow ve stavu *Ready*.
8. Po úspěšném nasazení je třeba se manuálně přihlásit do platformy MISP, čímž dojde k aktivování API klíče.
9. V případě potřeby využití některého z IntelOwl externích analyzátorů je dále třeba zadat odpovídající API klíč do webového rozhraní IntelOwl-u.
10. Pro prvotní synchronizaci komponenty pro CPE párování je nutné v rozhraní Apache Airflow spustit DAG *asset_sync_dag* pomocí tlačítka ve sloupci *Actions,* volba *Trigger dag*.

## 3.7. Ověření instalace

Ověření funkčnosti první a druhé části instalace se vykoná spuštěním dotazu z webového rozhraní aplikace Apache Superset:

1. Přihlášení se do aplikace Superset pomocí přihlašovacích údajů z tabulky níže.
2. V horní liště zvolit záložku „*SQL Lab -> SQL Editor“.*
3. Vlevo, v možnosti *Database* zvolit „*Trino (Iceberg)“.*
4. V možnosti *Schema* zvolit „*default“.*
5. Spustit následující dotaz.

```
$ SELECT count(*) from cz_muni_csirt_demo_ipfix_60_s;
```

6. V případě vrácení nějakého počtu jsou první dvě části plně funkční.

### Start orchestračních scénářů

**Cesta v adresáři výsledku**: `<helm-charts/orchestration-demo/data>`

Ověření celku se vykoná nahráním souboru obsahujícího phishingový email spolu s hlavičkami přes webové rozhraní *Phishing Use Case Uploader* a zadáním *CVE* identifikátoru ve službě *CVE Use Case Uploader*. Po těchto krocích by ve službě *Orion-Frontend* měly být vytvořené příslušné případy.

1. Otevřít službu *Phishing Use Case Uploader* na adrese dle tabulky níže.
2. Nahrát soubor s phishingovým mailem spolu s hlavičkami.
3. Ve službě *CVE Use Case Uploader* zadat „CVE-2022-3300“.
4. Přihlásit se do služby O*rion-Frontend* s údaji z tabulky níže.
5. V levém menu zvolit záložku „*Cases*“ a ověřit správně vytvořené případy dle nahraného phishingového souboru a zadaného CVE.

V následující tabulce se nachází přednastavené přístupové údaje k použitým službám.

| Služba | URL | Jméno | Heslo |
|---|---|---|---|
| Apache Airflow | orion-airflow.svc.orion.lan | admin | CHANGE_ME_123 |
| Apache Superset | orion-superset.svc.orion.lan | admin | CHANGE_ME_123 |
| CVE Use Case Uploader | orion-cve-uploader-frontend.svc.orion.lan | - | - |
| IntelOwl | orion-intelowl.svc.orion.lan | admin | CHANGE_ME_123 |
| Keycloak | orion-keycloak.svc.orion.lan | admin | CHANGE_ME_123 |
| Mattermost | orion-mattermost.svc.orion.lan | admin | CHANGE_ME_123 |
| Minio S3 | orion-minio.svc.orion.lan | admin | CHANGE_ME_123 |
| Misp | orion-misp.svc.orion.lan | admin@example.com | CHANGE_ME_123 |
| Netbox | orion-netbox.svc.orion.lan | admin | CHANGE_ME_123 |
| Orion-Frontend | orion-frontend.svc.orion.lan | admin | CHANGE_ME_123 |
| Phishing Use Case Uploader | orion-upload-frontend.svc.orion.lan | - | - |
| Trino | orion-trino.svc.orion.lan | admin | CHANGE_ME_123 |

## 3.8. Diagnostika

V následující sekci jsou popsané nejčastější problémy, které se mohou vyskytnout při instalaci a nasazení tohoto softwaru.

- Selhání při stahování obrazu kontejneru, Python knihovny nebo Helm závislosti. Je třeba zkontrolovat následující:
    - Existence a správnost přístupových údajů k registru, z něhož má být daný artefakt stažen.
    - Existence dané verze artefaktu v registru.
    - Jméno stahovaného artefaktu. Při některých chybách se může stát, že se použije přednastavená hodnota namísto Vámi specifikované.
    - V případě problémů se stažením obrazu kontejneru je taky vhodné zkontrolovat dostupnost registru z některého z uzlů v klastru.
- Neustálé selhávání některého z instalačních procesů (Superset, Airflow, …) – je třeba odstranit starou instanci instalace a spustit novou.

  ```
  $ helm uninstall <jméno_instalace> [-n namespace jméno_prostředí]
  $ helm install …
  ```

- Při použití self-signed certifikátů pro Ingress komponenty je třeba použít vlastní certifikační autoritu, podepisující dané certifikáty, a importovat ji do používaného webového prohlížeče. V případě, že tento krok nebude proveden, se budou stránky s webovým rozhraním zobrazovat jako nedůvěryhodné a muže dojít k chybě při zobrazovaní obrázku ve službě Orion Frontend.
- Nemožnost editace sekce SECURE_EXTRA při konfigurace služby Apache Superset.
    - Pozor na neviditelné znaky a redundantní mezery v kopírovaném textu.
- K aplikacím není připojováno připravené úložiště (volume).
    - V případě instalace transformace dát na klastr s vícerými uzly je nutné použít *StorageClass* podporující *ReadWriteMany* přístup. Tato funkce není využívána všemi službami z důvodu optimalizace výkonu, pro některé je ale její funkčnost kritická.
- Aplikace není dostupná na uváděných adresách:
    - Zkontrolovat logovací záznamy dotknutých služeb.
    - Zkontrolovat konfiguraci Ingress objektů.
    - Zkontrolovat, jestli jsou pody v *Ready* stavu, důvodů pro selhání kontroly *readiness* může být například nedostupná databáze.
- V případě, že se z nějakého důvodu nepodaří vytvořit uživatele pro službu IntelOwl, je nutné smazat všechny repliky podu *intelowl-chart-app*. Po smazání a opětovném nasazení Podu dojde ke korektnímu vytvoření uživatele a nastavení výchozího tokenu.

## 3.9. Přehled artefaktů a jejich typů

| Služba | Výsl. číslo | Python aplikace | Python knihovna | Spark / Angular | Docker | Helm |
|---|---|---|---|---|---|---|
| csirtmu-spark-iceberg-writer | 1 | | | X | | |
| csirtmu-asset-database-netbox-rest | 1 | X | | | X | X |
| csirtmu-case-federation | 1 | X | | | X | X |
| csirtmu-data-platform-service | 1 | X | | | X | X |
| csirtmu-observable-database | 1 | X | | | X | X |
| csirtmu-user-service | 1 | X | | | X | X |
| csirtmu-strawberry-dmt | 1 | | X | | | |
| docker-superset | 1 | | | | X | |
| csirtmu-apollo-gateway | 1 | | | | X | X |
| data-pipeline | 1 | | | | | X |
| data-platform | 1 | | | | | X |
| hive-metastore | 1 | | | | X | X |
| ipfixcol2 | 1 | | | | X | X |
| syslog-ng | 1 | | | | X | X |
| trino | 1 | | | | | X |
| generic-helm-chart | 1 | | | | | X |
| csirtmu-cpe-matching | 2 | | | | X | X |
| csirtmu-cve-use-case-backend | 2 | X | | | X | X |
| csirtmu-dns-firewall | 2 | X | | | X | X |
| csirtmu-exafs-mock-graphql-service | 2 | X | | | X | X |
| csirtmu-exafs-mock-service | 2 | X | | | X | X |
| csirtmu-observable-analyser-service | 2 | X | | | X | X |
| csirtmu-observable-parser-service | 2 | X | | | X | X |
| csirtmu-passive-dns-aggregator | 2 | X | | | X | X |
| csirtmu-phishing-use-case-backend | 2 | X | | | X | X |
| csirtmu-similarity-milvus | 2 | X | | | X | X |

| | | | | | | |
|---|---|---|---|---|---|---|
| apache-airflow-plugins/* | 2 | | X | | | |
| csirtmu-dataclass-config-loader | 2 | | X | | | |
| csirtmu-observable-analyser | 2 | | X | | | |
| csirtmu-observable-parser | 2 | | X | | | |
| airflow-triggerer | 2 | | | | X | |
| csirtmu-cve-use-case-frontend | 2 | | | X | X | X |
| csirtmu-phishing-use-case-frontend | 2 | | | X | X | X |
| airflow | 2 | | | | | X |
| csirtmu-dns-blocking-chart | 2 | | | | | X |
| intelowl-chart | 2 | | | | | X |
| janusgraph | 2 | | | | | X |
| misp-chart | 2 | | | | | X |
| orchestration-demo | 2 | | | | | X |
| csirtmu-auth-keycloak-fixture-loader | 3 | X | | | X | |
| csirtmu-case-database | 3 | X | | | X | X |
| csirtmu-integration-test-service | 3 | X | | | X | |
| csirtmu-service-gateway | 3 | X | | | X | X |
| csirtmu-ttp-database | 3 | X | | | X | X |
| csirtmu-user-profile-introspection-service | 3 | X | | | X | X |
| csirtmu-auth-keycloak-test-utils | 3 | | X | | | |
| csirtmu-auth-utils | 3 | | X | | | |
| csirtmu-starlette-jwt-auth-backend | 3 | | X | | | |
| csirtmu-orion-frontend | 3 | | | X | X | X |
| csirtmu-orion-microframework | 3 | | | X | | |

## 3.10. Příklady Kubernetes manifestů

### Příklad manifestu pro Longhorn StorageClass

```
---

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
    name: longhorn
provisioner: driver.longhorn.io
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: Immediate
parameters:
    numberOfReplicas: "2"
    staleReplicaTimeout: "2880"
    fromBackup: ""
    fsType: "ext4"
```

### Příklad manifestu pro Kubernetes ClusterIssuer

```
---

apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
    name: snakeoil-salesman
spec:
    selfSigned: { }
```

## 3.11. Příklad použití registru artefaktů v nástroji GitLab

**Python knihovna**

```
$ poetry config repositories.ORION https://gitlab.orion.lan/api/v4/projects
      /<id_projektu>/packages/pypi
$ poetry config http-basic.ORION token <heslo>
$ poetry publish --build -r ORION
```

**Python aplikace**

```
$ poetry source add ORION https://gitlab.orion.lan/api/v4/projects
      /<id_projektu>/packages/pypi
$ poetry config http-basic.ORION token <heslo>
$ poetry add --source ORION csirtmu-dataclass-config-loader
```

**Spark–Java**

```
$ mvn package jib:build -DskipTests=true \
  -Djib.to.auth.username=<jméno> -Djib.to.auth.password=<heslo> \
  -Djib.to.image=registry.gitlab.orion.lan:443/orion/orion-artifact-
repository/moje_aplikace:v1.0.0
```

**Docker**

```
$ docker login -u <jméno> -p <heslo> registry.gitlab.orion.lan:443
$ docker build -t registry.gitlab.orion.lan:443/orion/orion-artifact-
repository/moje_aplikace:v1.0.0 .
$ docker push registry.gitlab.orion.lan:443/orion/orion-artifact-
repository/moje_aplikace:v1.0.0
```

**Docker v případě projektových Python aplikací**

```
$ docker build -t moje_aplikace \
  --build-arg POETRY_HTTP_BASIC_ORION_USERNAME=token \
  --build-arg POETRY_HTTP_BASIC_ORION_PASSWORD=<heslo> .
```

**Helm**

```
$ helm package --dependency-update .
$ curl --request POST --user <jméno>:<heslo> --form "chart=@moje-aplikace-chart-
1.0.0.tgz \
https://gitlab.orion.lan/api/v4/projects/<id_projektu>/packages/helm/api/stable/c
harts
```

# 4. Uživatelský manuál

## 4.1. Návrhové koncepty uživatelského rozhraní

„Uživatelské rozhraní“ je rozděleno do několika agend. Agenda sdružuje pohledy dle jejich logických vazeb. Příkladem agendy může být správa případů. Pohledy jsou pak jednotlivé obrazovky aplikace. Každý pohled má svoji unikátní adresu v rámci aplikace. Příkladem pohledu může být přehled případů či analytický pohled konkrétního případu.

Během úvodní analýzy byly na základě očekávané komplexnosti a interaktivnosti definovány čtyři kategorie pohledů, jejichž konkrétní instance byly postupně realizovány v „Uživatelském rozhraní“.

- Obecný pohled – zobrazuje souhrnné a přehledové informace o různých konceptech a poskytuje tak všeobecný přehled o jejich množství a základním stavu.
- Detailní pohled – zobrazuje podrobné informace o stavu, historii a konkrétních parametrech daného konceptu, případně zobrazuje základní vazby na ostatní koncepty.
- Pokročilý pohled – specializované zobrazení dat souvisejících s konkrétním konceptem. Může zobrazovat pokročilé vazby mezi koncepty. Podpora základní editace.
- Analytický pohled – podpora pokročilé manipulace s daty souvisejících s konkrétním konceptem. Dovoluje vytvářet a aktualizovat vazby mezi koncepty.

Dále byly během úvodní analýzy definovány následující role potenciálních uživatelů relevantní pro „Uživatelské rozhraní“.

- Vedoucí CSIRT týmu,
- L1-L3 specialisté,
- správce ICT.

Vedoucí CSIRT týmu by především měl mít základní přehled o stavu ICT infrastruktury, počtu a stavu případů a úkolů. Z tohoto hlediska jsou pro roli vedoucího CSIRT týmu primárními agendami specializovaná agenda dashboard a správa přehledů, zejména pohled přehledu případů. Pro vedoucího CSIRT týmu může být užitečné sledovat i agendy aktiv.

Primární agendou pro role L1-L3 specialistů je agenda správy případů, zejména analytické pohledy na detaily případů a úkolů. Dále jsou pro ně užitečné všechny agendy, jejichž entity mají vazbu na případ či úkol, tedy agendy aktiv, zájmových údajů, TTP, IPFIX a evidence uživatelů.

Pro správce ICT je nejdůležitější zobrazení stavu spravované části infrastruktury a zobrazení přehledu síťového provozu. Z tohoto důvodu jsou pro správce ICT nejdůležitějšími agendami agenda aktiv a IPFIX.

Většinu pohledů můžeme členit i podle vztahu pohledu k entitám dané agendy do následujících kategorií.

- Rozcestník,
- přehled entit,
- detail entity,
- formulář.

Rozcestníkové pohledy jsou vhodné pro komplexní agendy sdružující mnoho přehledových nebo jiných pohledů. Rozcestníkové pohledy slouží jako domovská obrazovka dané agendy a nabízí uživateli graficky přívětivý přehled možností agendy a navigaci do dalších pohledů agendy. Užití tohoto typu pohledu není vhodné pro agendy, které jsou příliš jednoduché (příkladem může být agenda pozorovaných uživatelů) nebo jejich úvodním pohledem je logicky pohled přehledový (příkladem může být agenda správy případů).

Pohledy přehled entit slouží k zobrazení seznamu entit určitého typu. K vykreslení se převážně využívá komponenta tabulky, ale je možné využít i jiné komponenty, jako jsou karty nebo seznam. Pokud to aplikační rozhraní serveru dovoluje, jsou data vždy stránkována a mohou být filtrována i řazena podle

určitých atributů.  V takovém případě obsahuje pohled přehledu entit vysouvací panel filtrů a prvky uživatelského rozhraní pro ovládání řazení a stránkování. V případě nejčastějších tabulkových pohledů jsou tyto ovládací prvky integrovány do komponenty tabulky. Volitelně pak může přehledová obrazovka obsahovat jednotné ovládací prvky pro provádění základních operací nad entitou. Typicky se jedná o akce vytvoření, smazání, editace. Akce nesouvisející s konkrétní entitou jako je akce vytvoření či hromadné mazání, jsou zobrazeny v ovládacím panelu nad tabulkou. V případě akcí, které se vážou přímo ke konkrétní entitě, jako je například editace entity, jsou zobrazeny v odpovídajícím řádku tabulky. Navigace na pohled detailu entity je provedena kliknutím na buňku tabulky zobrazující jméno či identifikátor entity. V případě změn stránkování, filtrování či řazení, je stav uložen do URL jako dotaz (query params). V případě znovuotevření této URL je stav pohledu obnoven dle hodnot parametrů.

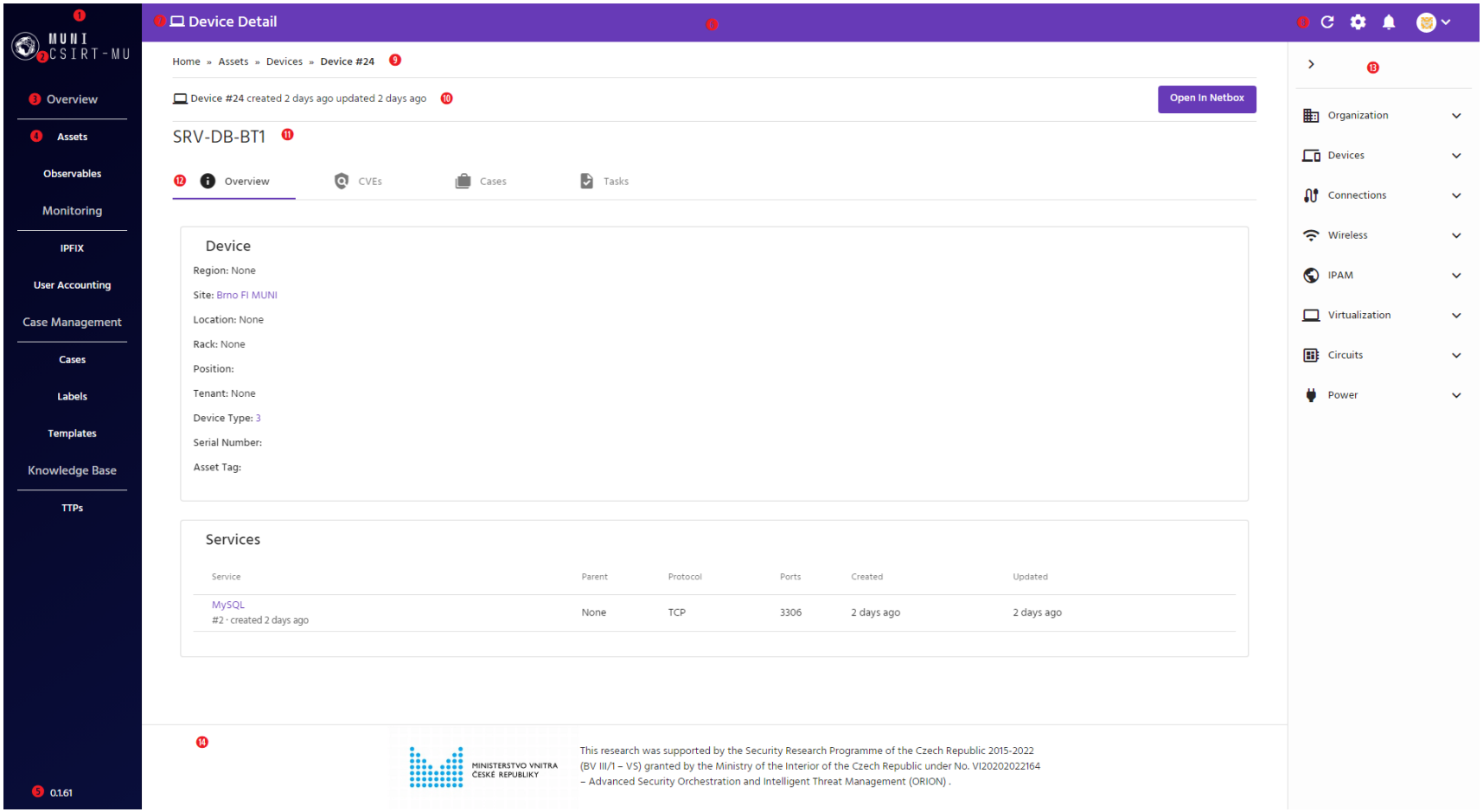
Pohledy na detail entity zobrazují konkrétní entitu. Může se jednat o pohledy velmi rozmanité komplexity, od obecných pohledů, zobrazující základní atributy konkrétní entity, až po vysoce interaktivní analytické pohledy. Typický pohled detailu entity obsahuje název, základní atributy entity jako je datum a čas vytvoření, identifikátor. Pokud detailní pohled obsahuje různé varianty pohledu, jsou jednotlivé varianty zobrazovány v přepínacích kartách. Komplexnější komponenty uvnitř přepínacích karet jsou zobrazovány ve vysouvacích panelech. Příkladem varianty může být zobrazení vazeb na případy v detailním pohledu zájmového údaje, příkladem komplexní komponenty může být komponenta pro správu souborů v detailním pohledu případu. Pokud entita podporuje editaci, může pohled přepínat mezi zobrazovacím módem a editačním módem. V případě změny aktivní přepínací karty, nebo editačního módu je stav uložen do URL jako dotaz. V případě znovuotevření této URL je stav detailní obrazovky obnoven dle hodnot parametrů.

Formulářové pohledy slouží k vytváření či editaci entit. Každý formulář obsahuje validaci a jednotlivá formulářová pole upozorňují na případné chyby ve validaci odpovídající chybovou hláškou, nebo nápovědou. Pokud formulář obsahuje neuložené změny a uživatelem je vyvolána navigace, která by mohla zavinit ztrátu dat ve formuláři, zobrazí se potvrzovací dialogové okno.

Celá aplikace a její pohledy jsou obaleny komponenty sdíleného rozložení z knihovny Sentinel Layout. Samotné pohledy maximálně využívají sdílené komponenty, služby a utility z knihovny Sentinel Components a mikro-frameworku *ORION* SCMFM.

## 4.2. Rozložení prvků uživatelského rozhraní

1. Postranní navigační panel
2. Logo
3. Kategorie agend
4. Agenda
5. Verze
6. Panel nástrojů
7. Název pohledu
8. Akce panelu nástrojů
9. Drobečková navigace
10. Panel nástrojů pohledu
11. Název entity
12. Přepínací karty
13. Boční kontextový panel
14. Zápatí




*Obrázek 17. Příklad pohledu uživatelského rozhraní*

## 4.3. Agendy

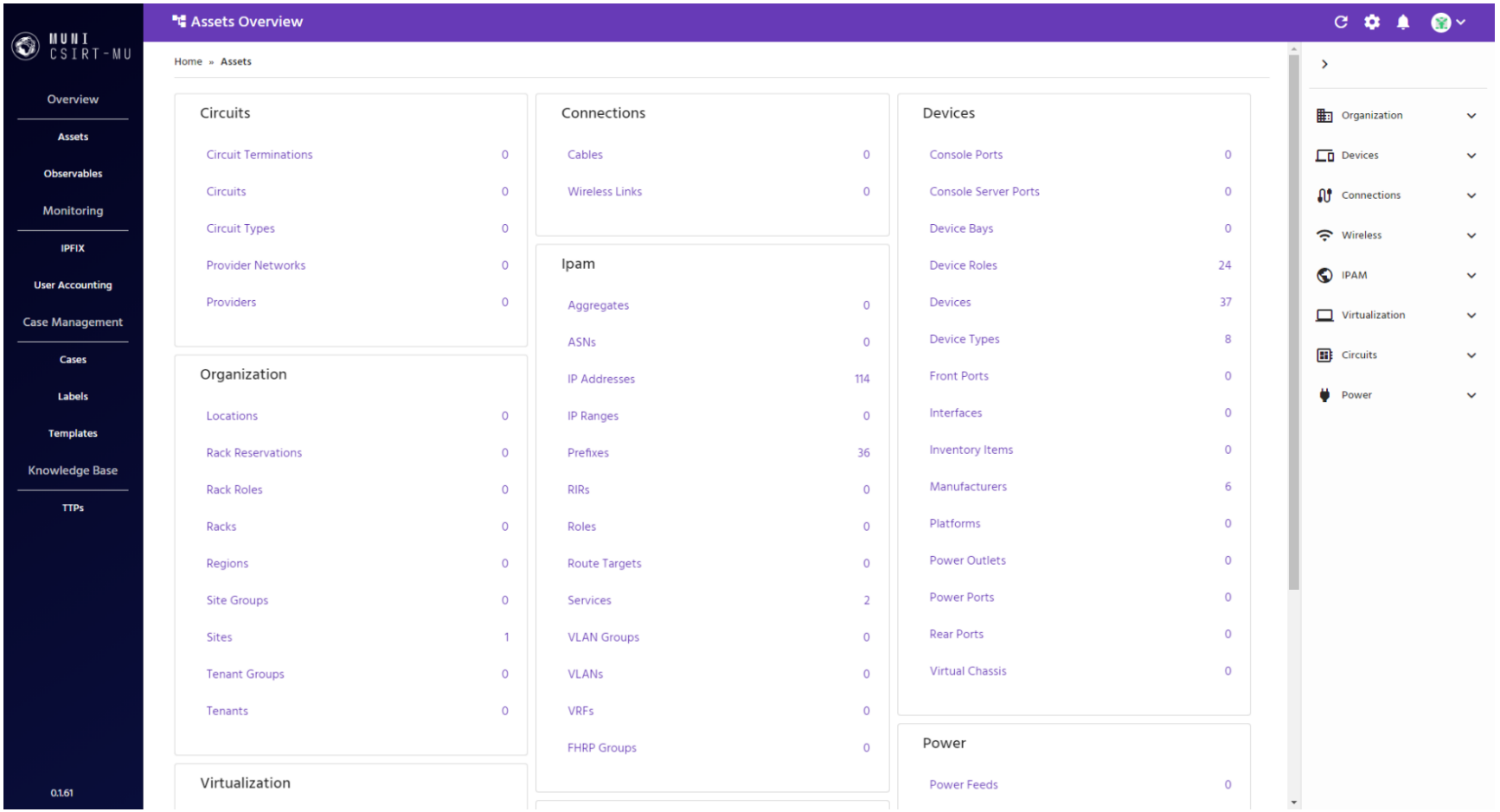
„Uživatelské rozhraní“ je rozděleno do několika agend. Agendy jsou logické celky spojující vzájemně provázané součásti aplikace. Každé agendě typicky náleží příslušná mikro-služba na straně back-end. Jednotlivé agendy jsou detailněji popsány v následujících sekcích.

### Agenda aktiv

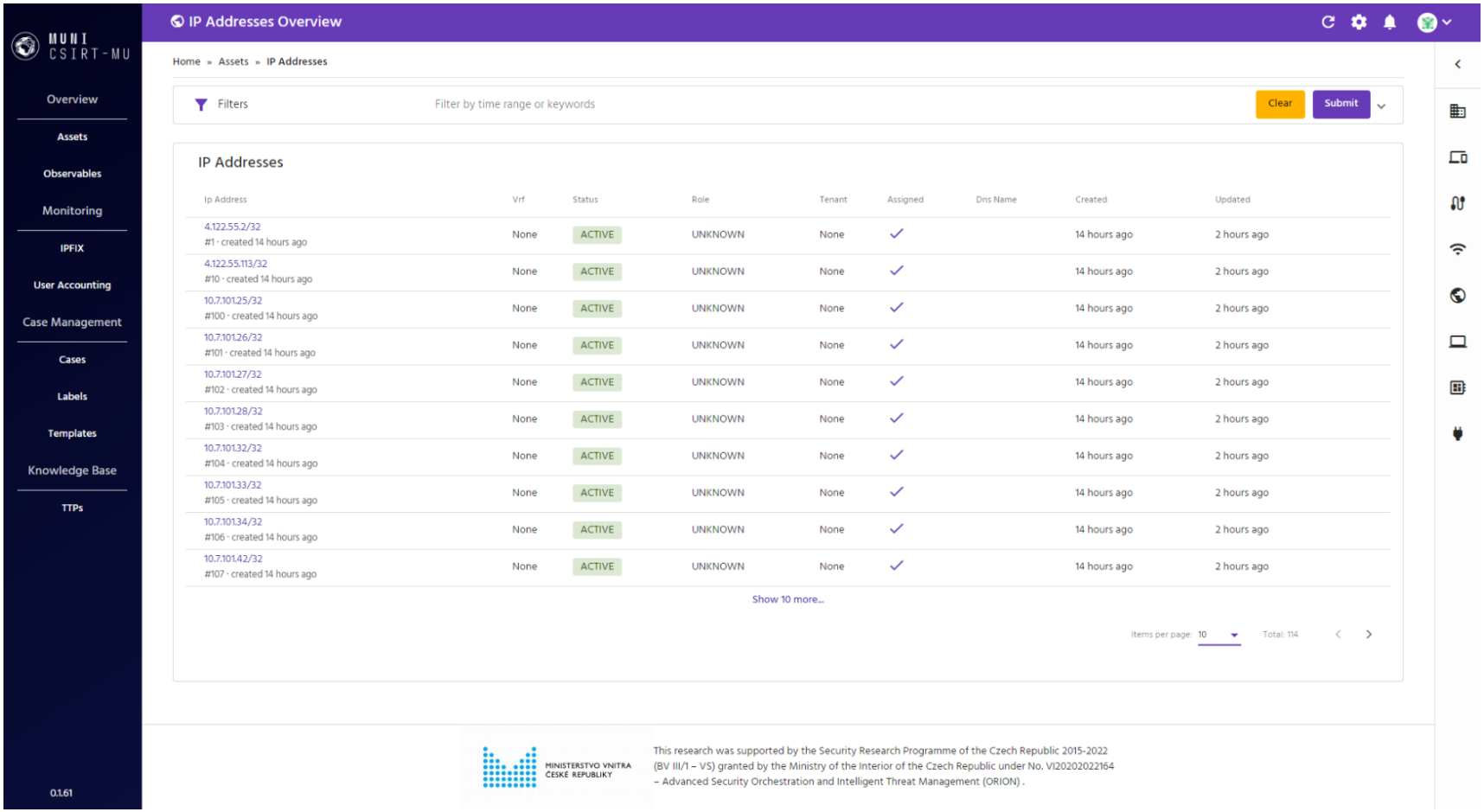
Agenda aktiv je postavena nad mikro-službou „Databáze aktiv“ a aplikací Netbox, která slouží k modelování infrastruktury. Datový model Netboxu rozšiřuje o CVE identifikátory a vztahy s případy a úkoly. Model obsahuje řadu typů, které lze zařadit do následujících kategorií.

- Organizace (organization),
- zařízení (devices),
- spojení (connections),
- bezdrátové sítě (wireless),
- IPAM,
- virtualizace (virtualization),
- obvody (circuits),
- napájení (power).

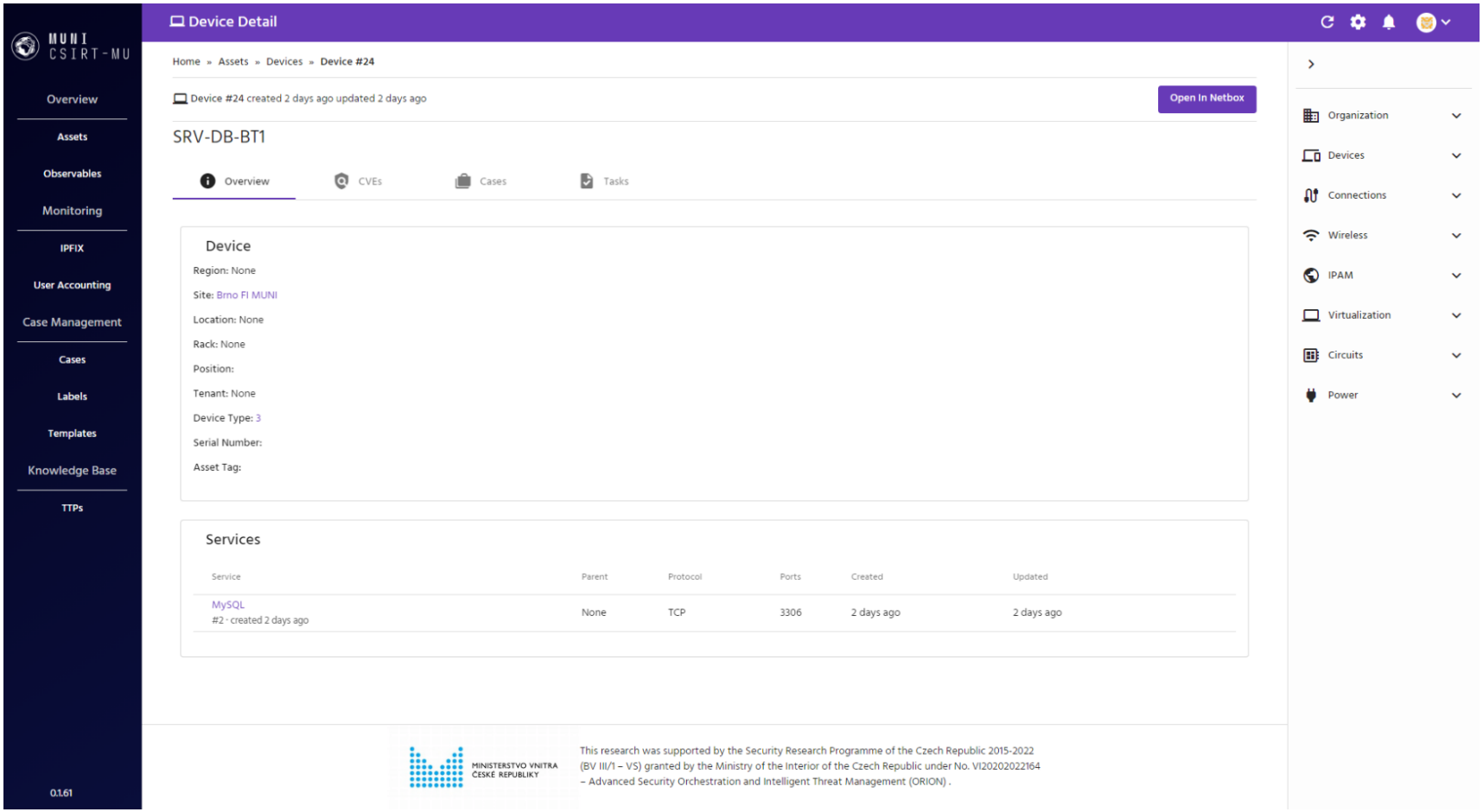
Z hlediska „Uživatelského rozhraní“ se agenda skládá z řady obecných a detailních pohledů. Rozcestníkový pohled zobrazuje přehled typů aktiv. Pro jednotlivé typy aktiv existují obecné a detailní pohledy. Rozcestníkový pohled obsahuje navigaci na obecné pohledy jednotlivých typů aktiv. Obecné pohledy vypisují aktiva daného typu ve formě tabulky se stránkováním. Tyto pohledy také podporují jednoduché filtrování. Detailní pohledy zobrazují základní informace o vybraném aktivu a vztahy s dalšími aktivy. V jednotlivých přepínacích kartách jsou pak formou tabulek zobrazeny jejich CVE, vztahy s případy a úkoly. Agenda slouží k zobrazení aktiv v režimu pouze ke čtení, nicméně každý detailní pohled obsahuje odkaz do uživatelského rozhraní aplikace Netbox, ve které je možné aktiva editovat a vytvářet.




*Obrázek 18. Obecný pohled agendy aktiv zobrazující rozcestník*




*Obrázek 19. Obecný pohled agendy aktiv zobrazující přehled aktiv typu IP adresa*




*Obrázek 20. Detailní pohled agendy aktiv zobrazující zařízení*

## Agenda zájmových údajů

Agenda slouží k zobrazení zájmových údajů, jejich CTI záznamů a vazeb na případy a úkoly. Datový model obsahuje několik typů zájmových údajů, mezi které patří IP adresa, doménové jméno, URL adresa, překlad a reverzní překlad doménového jména a IP adresy, emailová adresa, emailová zpráva.

„Uživatelské rozhraní“ zobrazuje zájmové údaje v režimu ke čtení, skládá se z obecných a detailních pohledů. Agenda obsahuje úvodní pohled, který vypisuje všechny zájmové údaje v tabulce podporující stránkování. Pohled zároveň obsahuje jednoduché filtrování. Dále má agenda detailní pohledy pro každý typ zájmového údaje, které zobrazují základní informace o zájmových údajích a jejich vztazích s dalšími zájmovými údaji. V jednotlivých přepínacích kartách jsou pak formou tabulek zobrazeny CTI záznamy, vztahy s případy a úkoly

## Agenda IPFIX

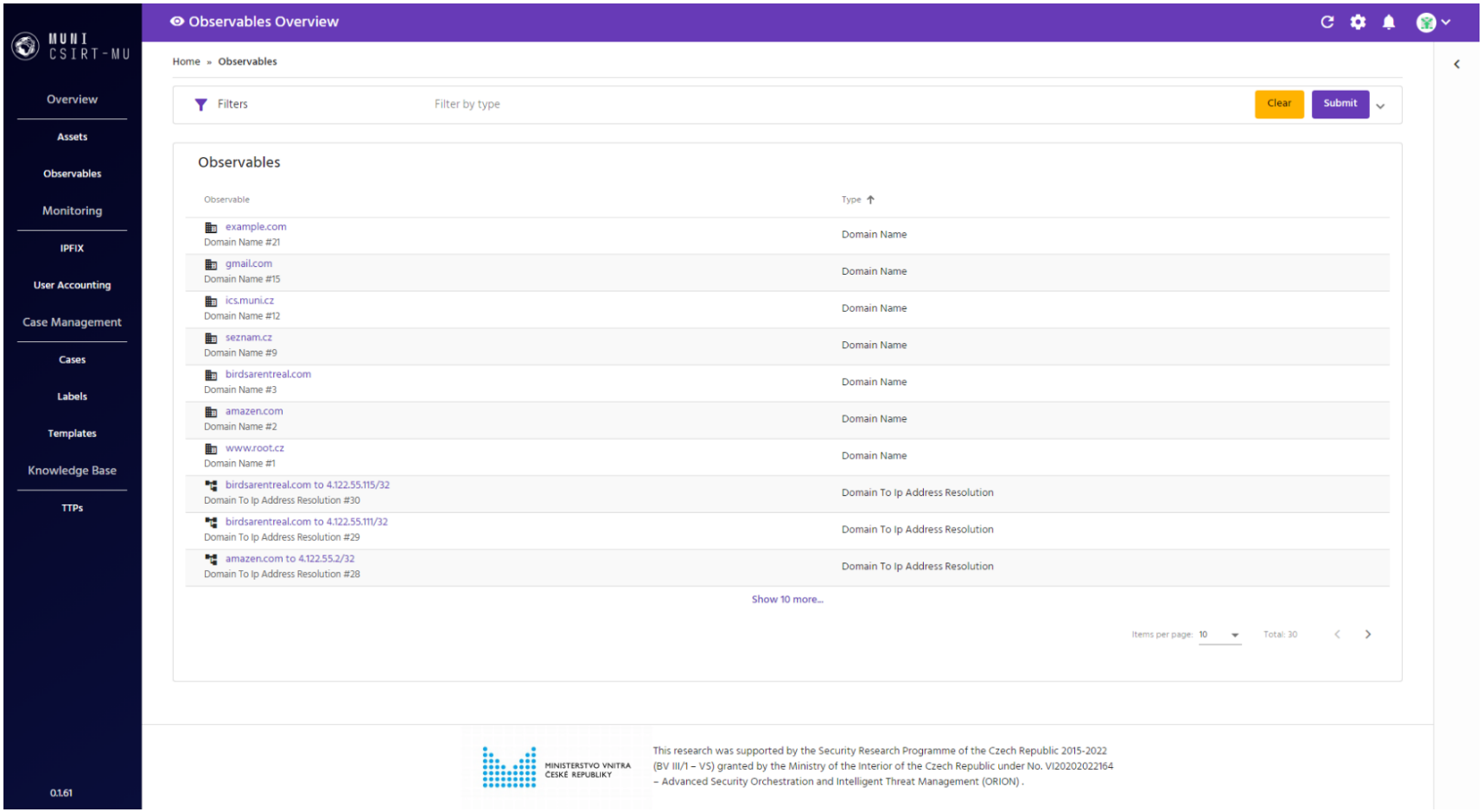
Agenda IPFIX patří k rozsahově menším. Uživateli dává možnost procházet záznamy síťových toků infrastruktury ve formátu IPFIX, řadit a filtrovat je dle IP adresy, portu, či časového rozsahu.

„Uživatelské rozhraní“ zobrazuje informace o síťových tocích formou detailního pohledu. Toky jsou zobrazeny v tabulce s filtrováním dle časového rozsahu a možností zobrazit detailní informace síťového toku přímo v rozbalovacím detailu řádku tabulky.

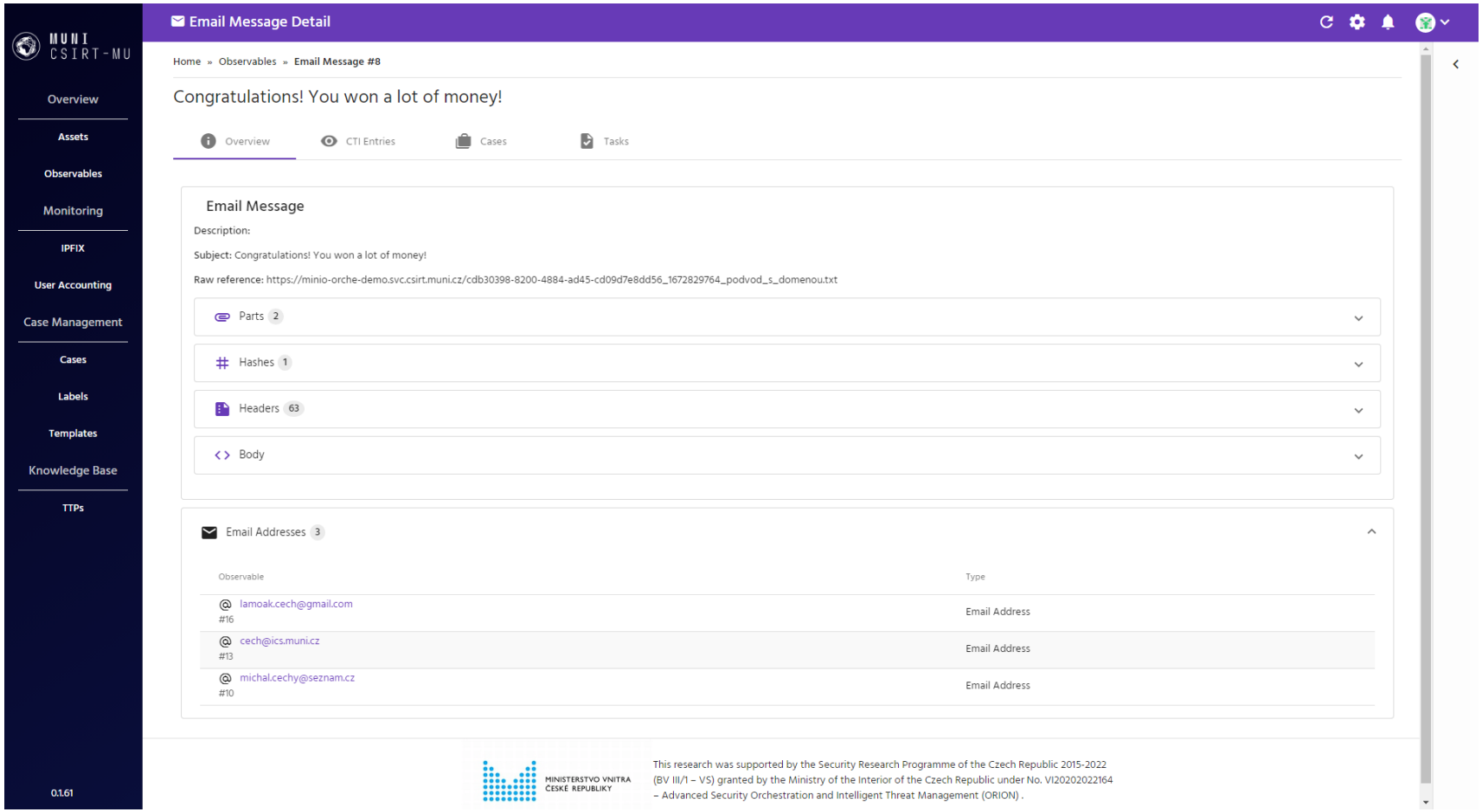
## Agenda evidence uživatelů

Evidence uživatelů je druhou agendou z kategorie monitoring. Jejím cílem je zobrazit identifikované uživatele a IP adresu, která jim byla v určitém časovém okně přiřazena. Agenda také zobrazuje všechny zaznamenané síťové toky, ve kterých IP adresa figurovala jako příjemce, nebo odesílatel.

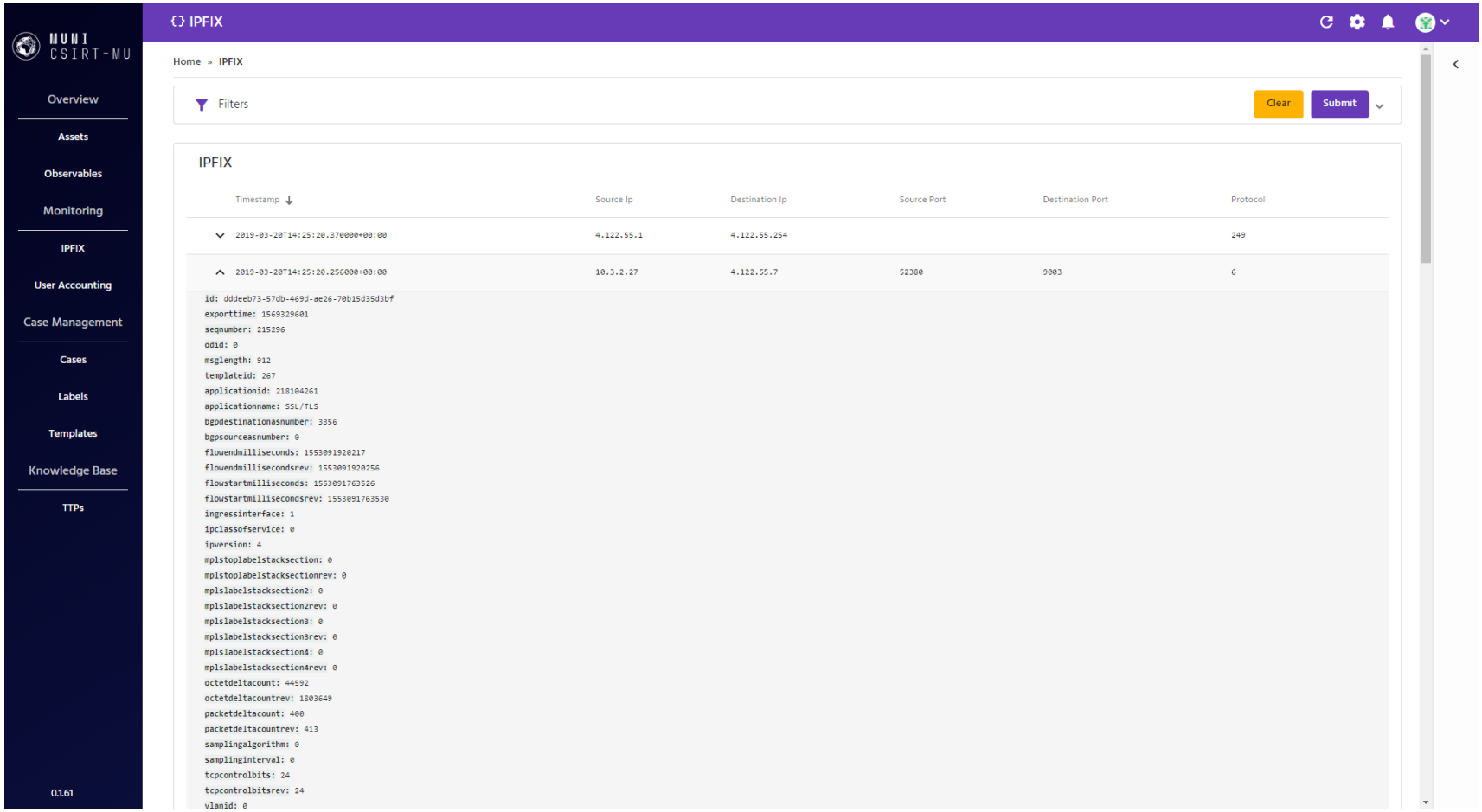
„Uživatelské rozhraní“ obsahuje několik pohledů detailní kategorie. Pohledy zobrazují data ve formě stránkované tabulky. Síťové toky k dané IP adrese je možné zobrazit v rozbalovacím detailu řádku tabulky. Kliknutím na uživatele, nebo IP adresu je navigováno na detailní pohled zobrazující všechna přiřazení vybraného uživatele k IP adresám, resp. přiřazení vybrané IP adresy k různým uživatelům.



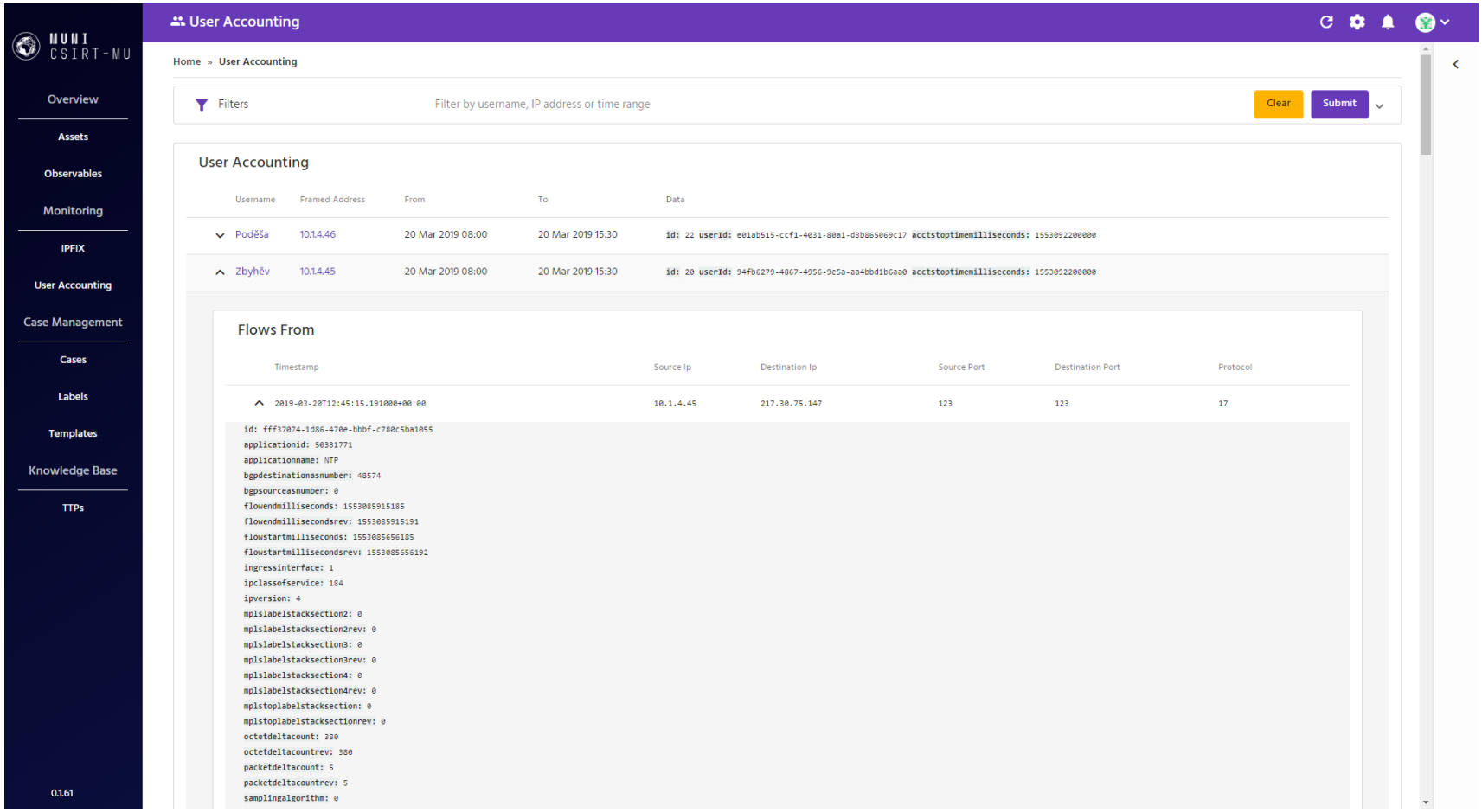

*Obrázek 21. Obecný pohled agendy zájmových údajů zobrazující přehled zájmových údajů*




*Obrázek 22. Detailní pohled agendy zájmových údajů zobrazující emailovou adresu*




*Obrázek 23. Detailní pohled agendy IPFIX zobrazující výpis síťových toků s rozbaleným detailem řádku tabulky*




*Obrázek 24. Detailní pohled agendy evidence uživatelů zobrazující přehled mapování uživatelů na IP adresy. V rozbaleném detailu řádku tabulky jsou vypsány síťové toky dané IP adresy*

## Agenda případů

Agenda případů je nejdůležitější agendou „Uživatelského rozhraní“. Jejím účelem je správa případu a všech jeho celků po dobu jeho celého životního cyklu. Zobrazuje stav průchodu případem formou pracovních postupů a fází pracovního postupu. Případ je řešen plněním jednotlivých úkolů, rovněž obsažených v této agendě. Komunikaci mezi řešiteli a určitou formu historie případu zajišťuje komentářová sekce případu a jeho úkolů.

Z hlediska „Uživatelského rozhraní“ agenda obsahuje obecný pohled na přehled případů, zobrazený formou stránkované tabulky s pokročilými možnostmi filtrování. Dále obsahuje vysoce interaktivní analytické pohledy na konkrétní případ, pracovní postup, fázi pracovního postupu a úkol. Všechny tyto entity je možné vytvořit pomocí formulářový pohledů. Analytické pohledy zobrazují poměrně velké množství atributů a vztahů dané entity. Zároveň umožňují tyto atributy a vztahy vytvářet, editovat a odstraňovat.

Tyto komplexní analytické pohledy jsou členěny do postranního kontextového panelu a hlavní části. Postranní panel zobrazuje základní editovatelné atributy entity, jako jsou řešitelé, štítky, či priorita. Na vrcholu hlavní časti se nachází pracovní panel, obsahující základní informace o entitě, jako jsou její autor, datum vytvoření a stav. Zbytek prostoru hlavní části zabírají přepínací karty. Atributy a vztahy entity jsou zobrazeny ve vysouvacích panelech hlavní přepínací karty. Vysouvací panely jsou následovány komentářovou sekcí. V dalších přepínacích kartách se pak nachází vztahy s různými entitami (aktiva, zájmové údaje, TTP, uživatelé, události, externí reference). Pomocí vztahů v přepínacích kartách je možno navigovat do ostatních agend „Uživatelského rozhraní“, a procházet tak grafem vazeb komplexních případů. Přepínací karty podporují základní filtrování a v některých případech i přidávání nových vztahů.

Komentáře a popisky ve všech analytických pohledech podporují formátování a vykreslení textu ve značkovacím jazyce Markdown.

## Agenda štítků

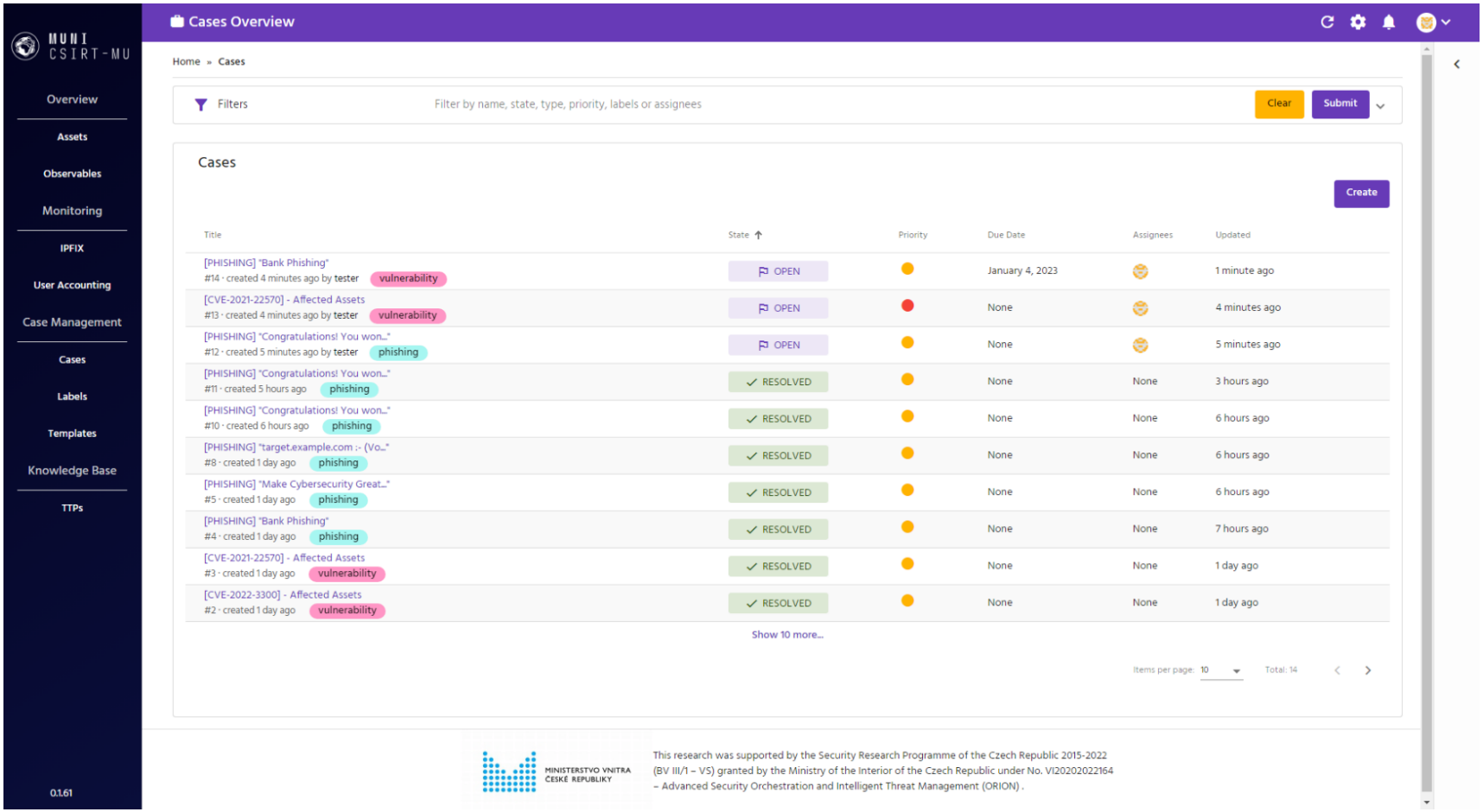
Agenda štítků je doplňkovou agendou ke správě případů. Obsahuje přehled a operace vytvoření, editace a odstranění štítků. Štítky mohou být přiřazovány případům či úkolům. Přiřazený štítek je zobrazen v přehledových i detailních pohledech na případy i úkoly a plní funkci výrazného, konfigurovatelného grafického prvku, který může sloužit ke kategorizaci či ohodnocení určité metriky definované týmem řešící případy. Případy je možné filtrovat dle přiřazených štítků.

„Uživatelské rozhraní“ obsahuje pokročilé pohledy sloužící k výpisu existujících štítků, vytváření a editaci štítků.

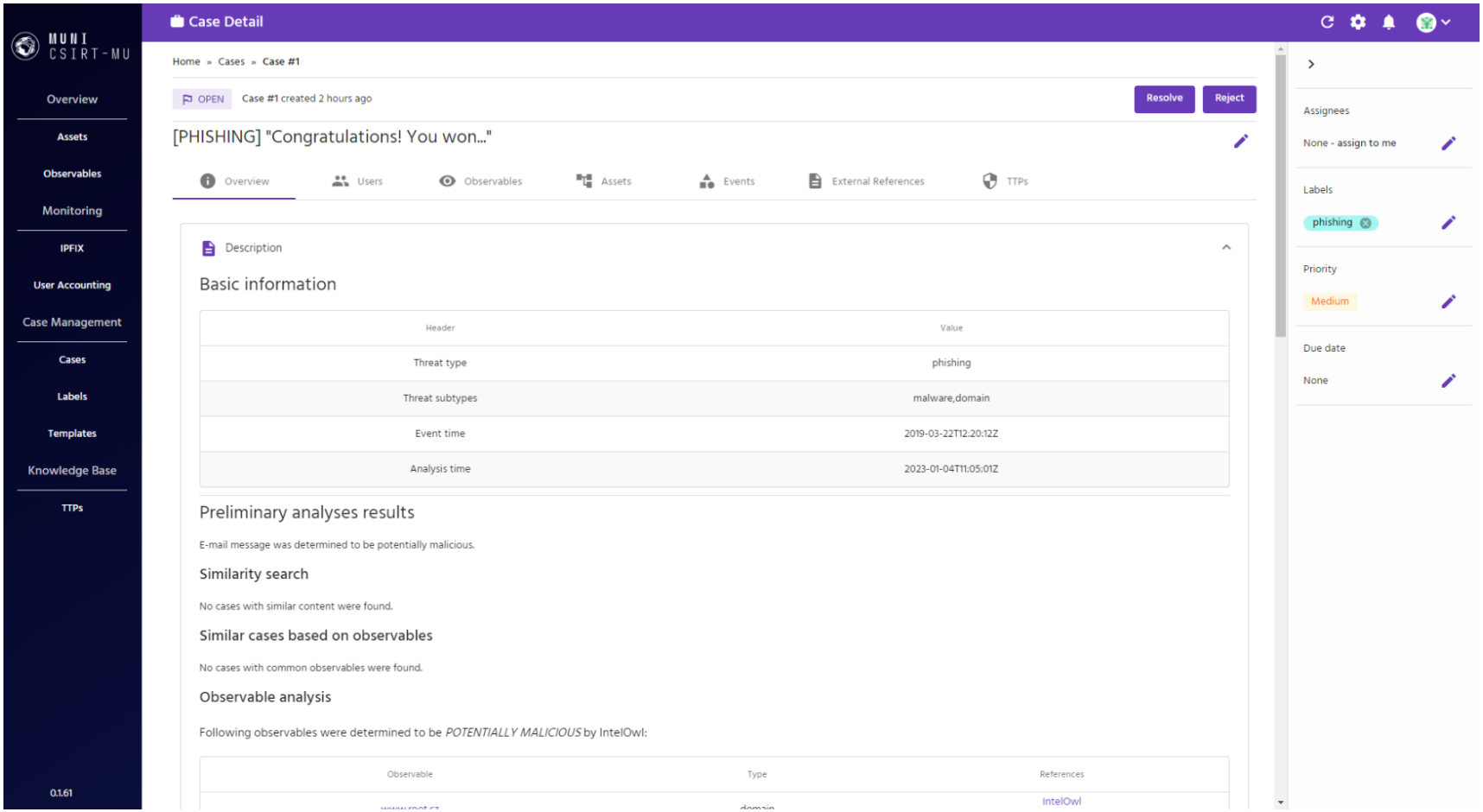
## Agenda šablon pracovního postupu

Další doplňkovou agendou je agenda šablon pracovního postupu. Šablony je v agendě možné nahrát či stáhnout do souboru ve formátu JSON, vytvořit je, editovat a odstranit. Hlavní agenda správy případů může šablony využívat k vytváření pracovních postupů. V takovém případě je pracovní postup, jeho fáze a úkoly vytvořeny dle parametrů popsaných v šabloně.

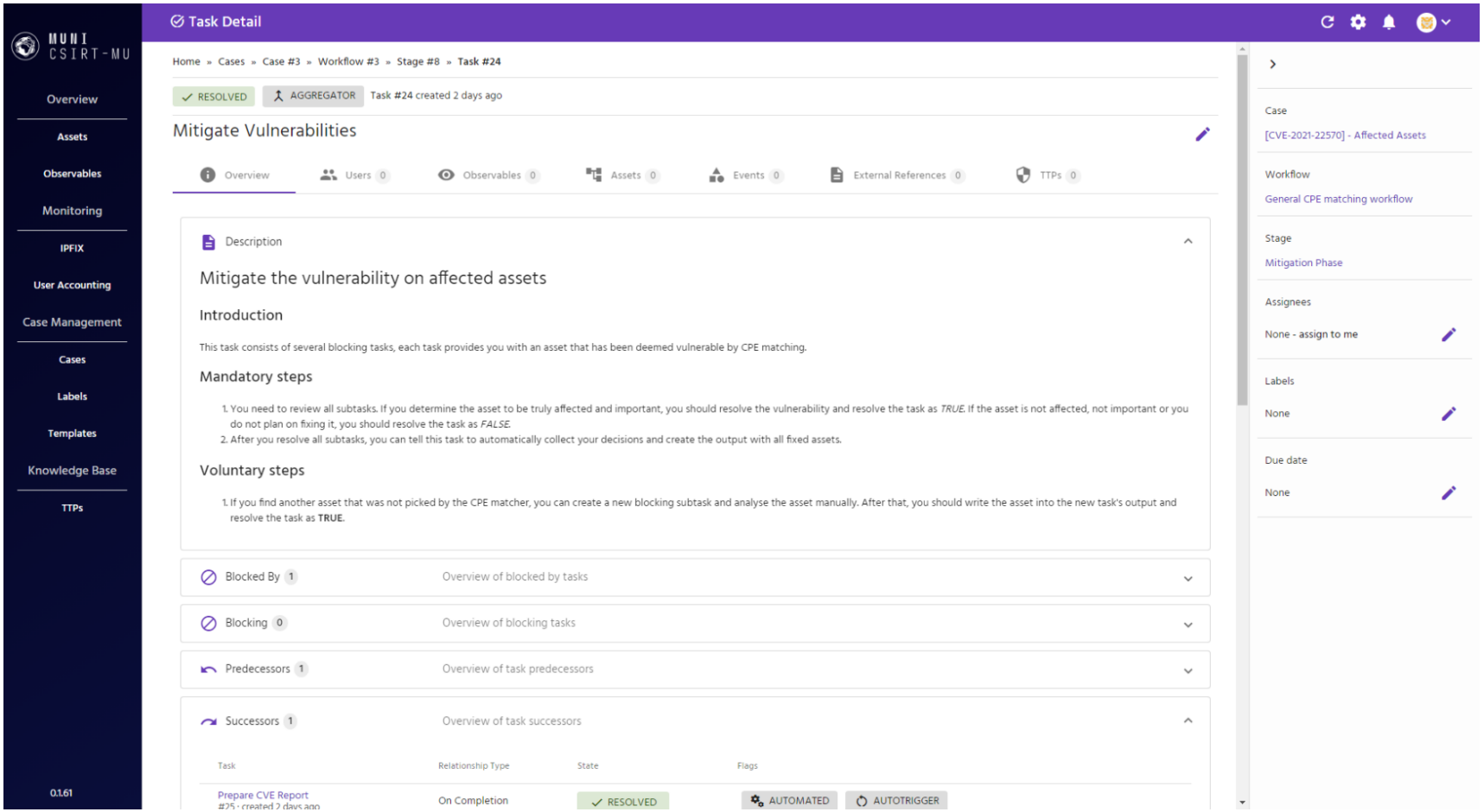
„Uživatelské rozhraní“ obsahuje pokročilé pohledy sloužící k výpisu existujících šablon, k vytváření a editaci šablon.



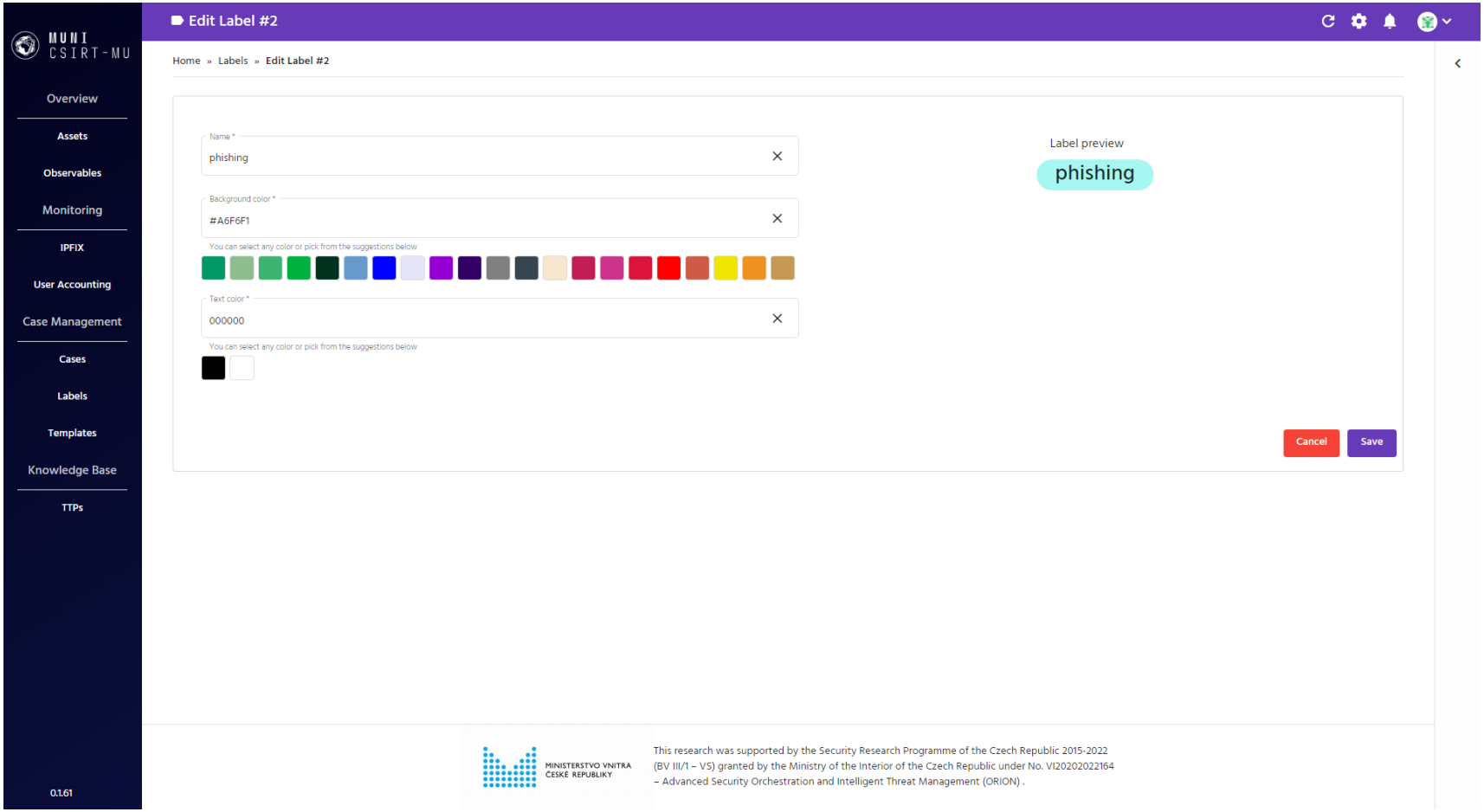

*Obrázek 25. Obecný pohled agendy správy případů zobrazující přehled případů*



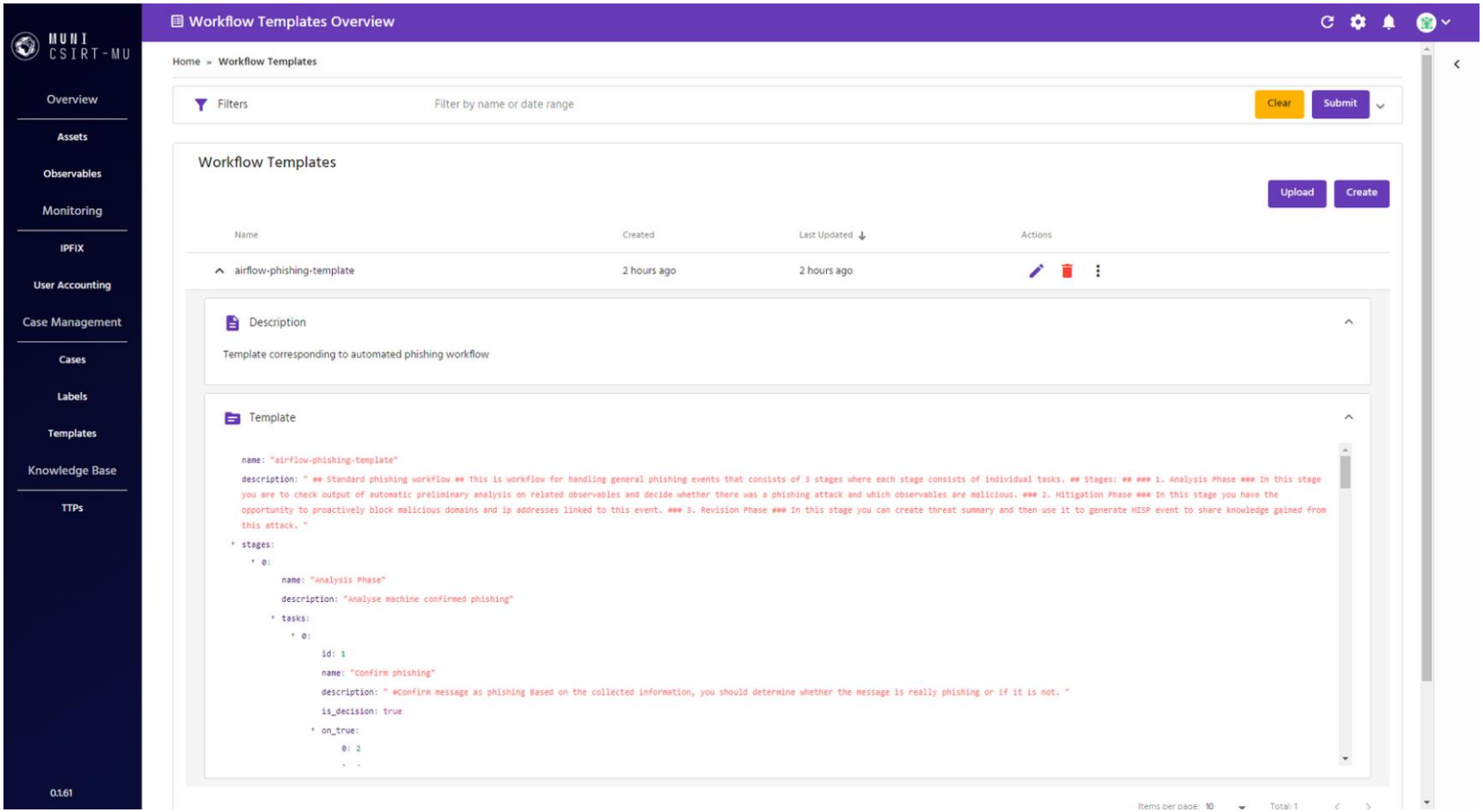

*Obrázek 26. Analytický pohled agendy správy případů zobrazující detail případu phishingu*




*Obrázek 27. Analytický pohled agendy správy případů zobrazující detail úkolu mitigace zranitelností aktiv*




*Obrázek 28. Pokročilý pohled agendy správy štítků zobrazující editaci štítku*




*Obrázek 29. Pokročilý pohled agendy správy šablon pracovního postupu. Zobrazuje přehled šablon s rozbaleným detailem řádku tabulky*

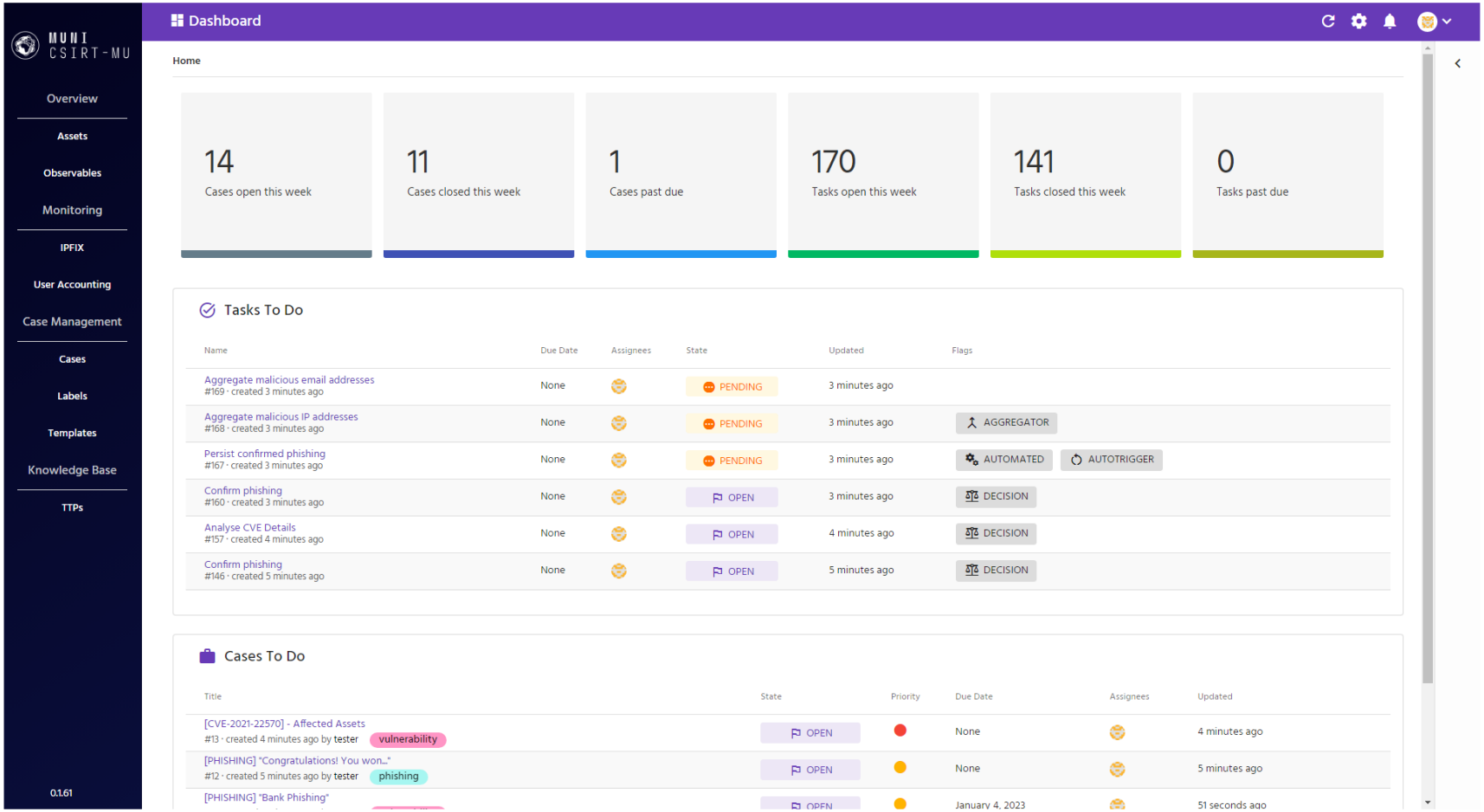
## Agenda přehledové obrazovky (Dashboard)

Dashboard je specializovaná agenda obsahující detailní pohled na aktuální stav řešení případů a úkolů. V horní části obrazovky se nachází několik vizualizačních prvků, zobrazujících základní přehledové údaje o množství případů a úkolů, které byly v časovém okně posledního týdne otevřeny či zavřeny. Zvýšená pozornost by měla být věnována vizualizačnímu prvku, který zobrazuje informaci o případech či úkolech, které přesahují požadovanou lhůtu dokončení. Ve spodní části obrazovky se pak nachází personalizované přehledy případů a úkolů formou seznamu „To-Do“. Seznam obsahuje případy a úkoly jež má právě přihlášený uživatel přiřazeny k řešení. Úkolem tohoto prvku „Uživatelského rozhraní“ je poskytnout (vedoucím) členům bezpečnostního týmu zjednodušený pohled na případy a úkoly vyžadující jejich pozornost. Podobně jako u vizualizačních prvků je v seznamu zvýrazněno, pokud je některý z přiřazených případů a úkolů nevyřešen po požadované lhůtě dokončení.

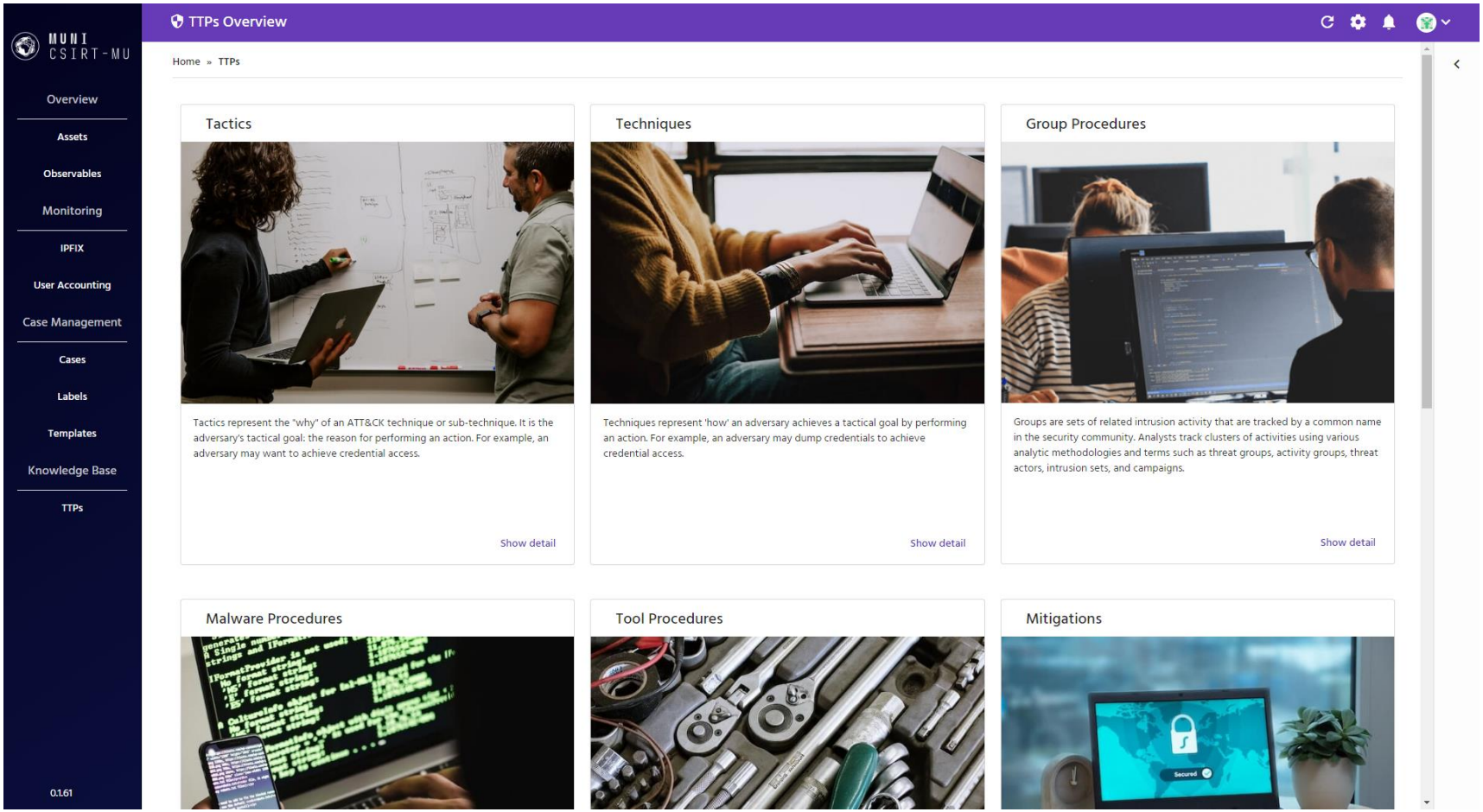
## Agenda TTP

Agenda TTP spadající do kategorie znalostní báze vychází z modelu MITRE ATT&CK®, který rozšiřuje a některé jeho prvky upravuje. Tato agenda slouží jako přehled známých znalostí o taktikách a technikách útočníků, nástrojích, platformách, malwaru, mitigacích a organizovaných skupinách. Jednotlivé entity znalostní báze taktéž obsahují vazby na existující případy a úkoly, se kterými mají nějakou souvislost. Tento vztah je oboustranný a z pohledu řešitele případu tak slouží nejen k načerpání znalostí o pojmech souvisejících s řešeným případem, ale i jako sdružení různých případů a úkolů s určitou mírou podobnosti k tomu právě řešenému.

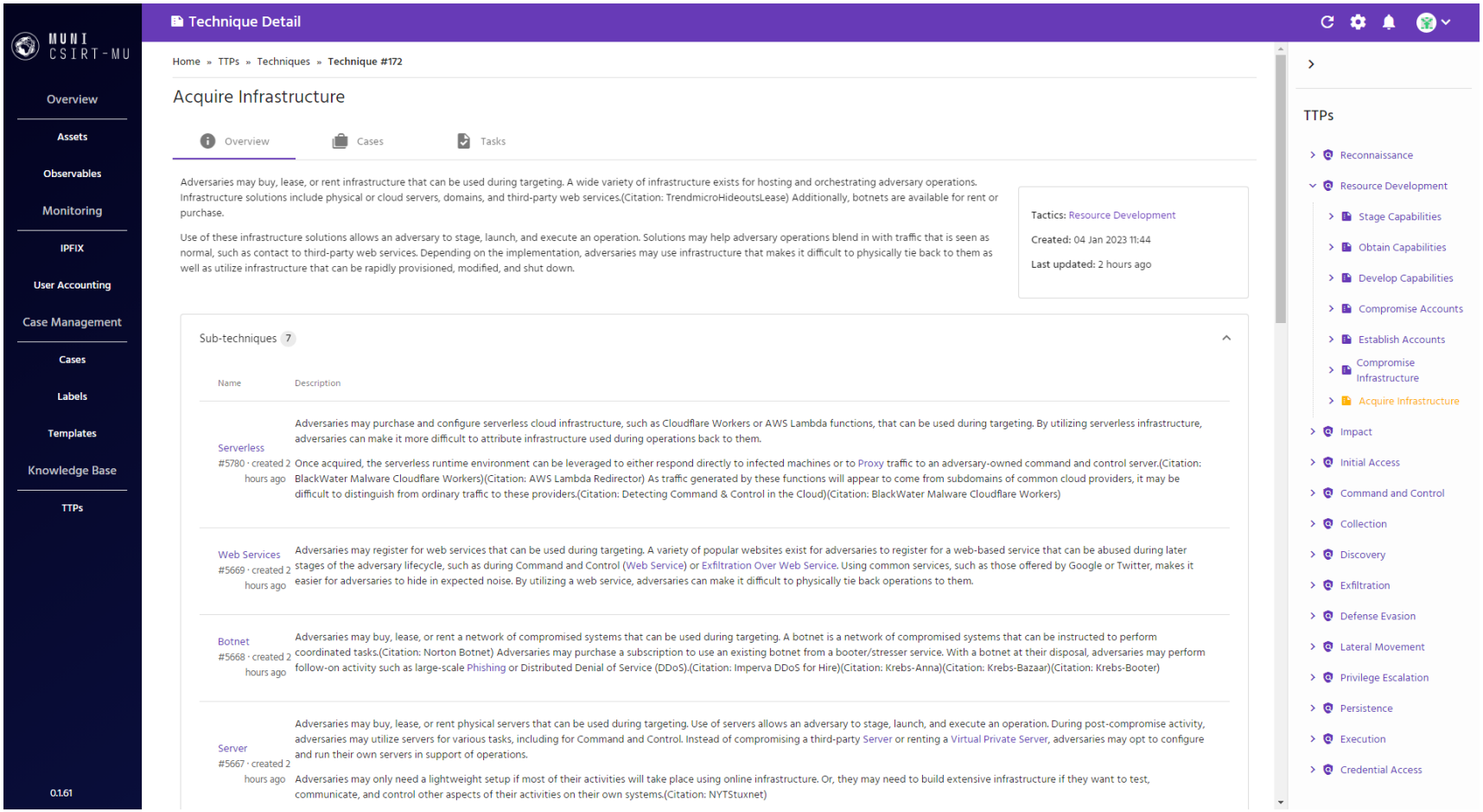
„Uživatelské rozhraní“ agendy TTP se skládá z řady obecných a detailních pohledů. Její úvodní pohled v několika kartách zobrazuje a popisuje nejdůležitější kategorie znalostní báze. Slouží rovněž jako rozcestník na pohledy jednotlivých kategorií, které typicky obsahují přehledovou tabulku s jednoduchým filtrováním. Prvky tabulky obsahují krátké popisky daných entit agendy a odkazy na jejich detailní pohledy různých typů. Jednotlivé pohledy pro dané typy jsou si poměrně podobné a obsahují podrobný popis prvku znalostní báze, vztahy k dalším entitám a vazby na případy a úkoly. V případě detailních pohledů na entity typu taktika či technika je pohled obohacen o boční navigační panel, který formou stromové hierarchie zobrazuje přehled všech taktik, technik a pod-technik.




*Obrázek 30. Speciální agenda dashboard*




*Obrázek 31. Obecný pohled agendy TTP zobrazující rozcestník TTP*



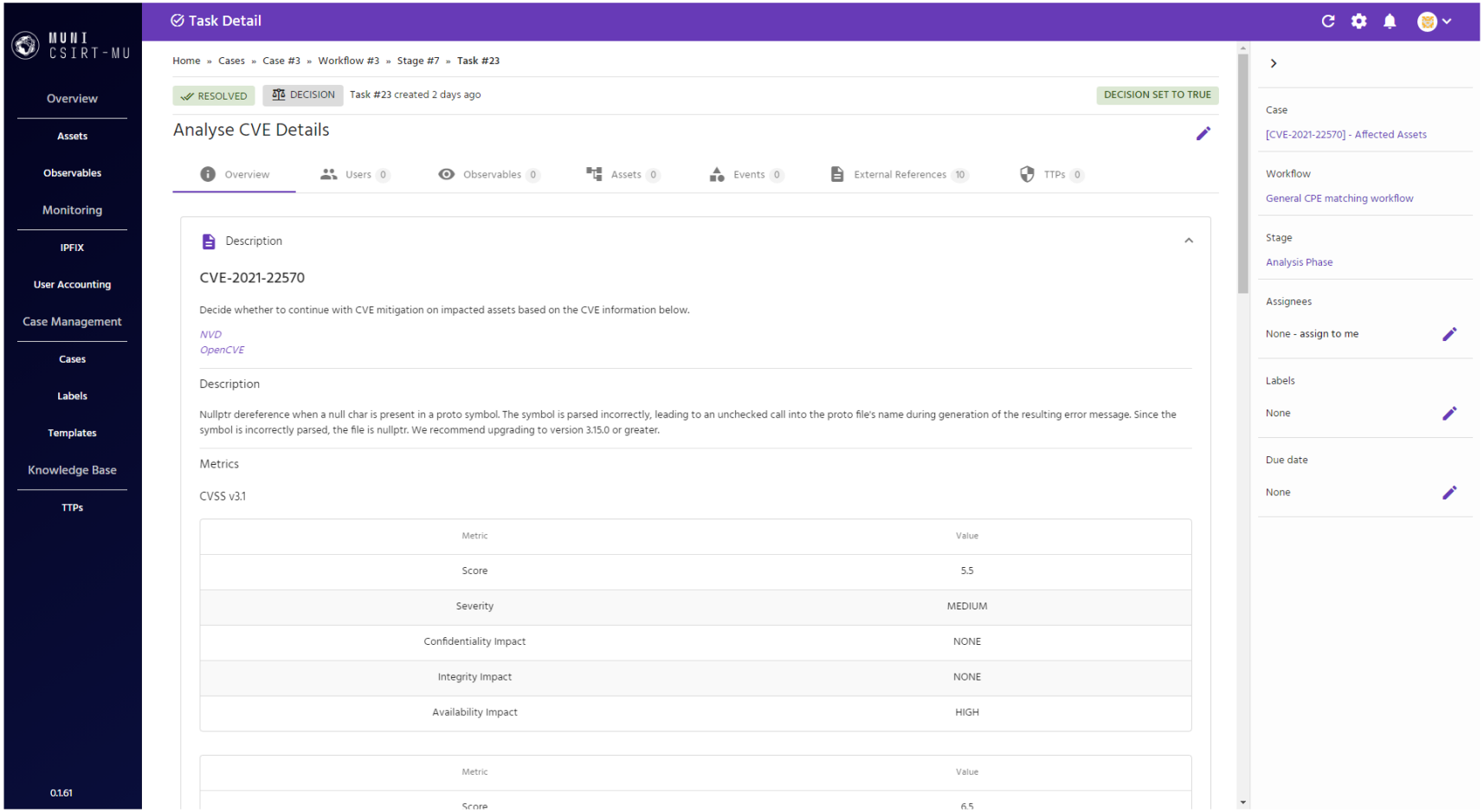

*Obrázek 32. Detailní pohled agendy TTP. Zobrazuje konkrétní techniku a její pod-techniky*

## 4.4. Demonstrační scénáře

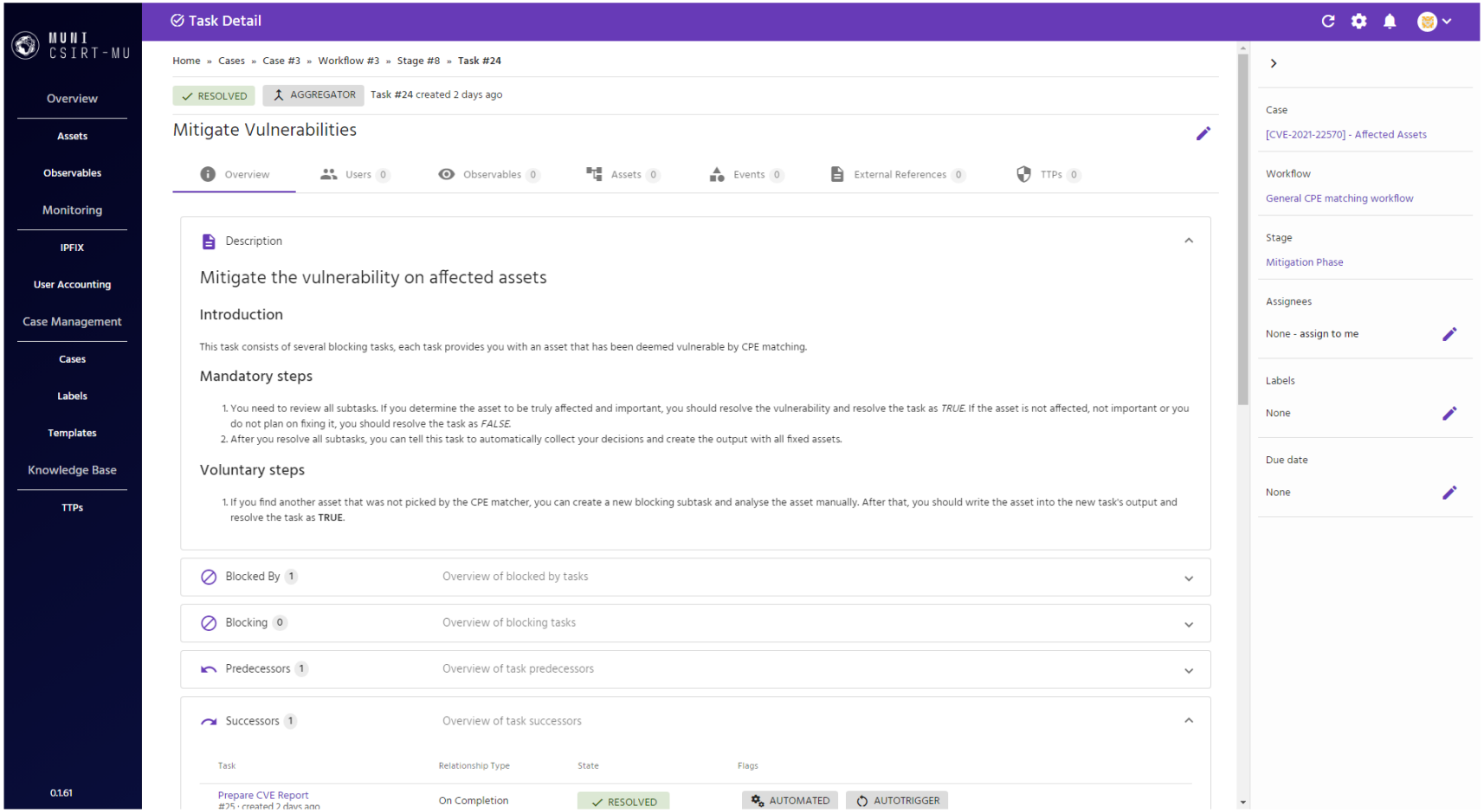
### Demonstrace řešení případu hrozby typu Zneužití zranitelnosti

Případ hrozby typu „Zneužití zranitelnosti“ je praktickým využitím „Orchestračního scénáře pro hrozby typu Zneužití zranitelnosti“ z Výsledku č. 2 [1]. Následuje popis jednotlivých kroků řešení případu v kontextu „Uživatelského rozhraní“.

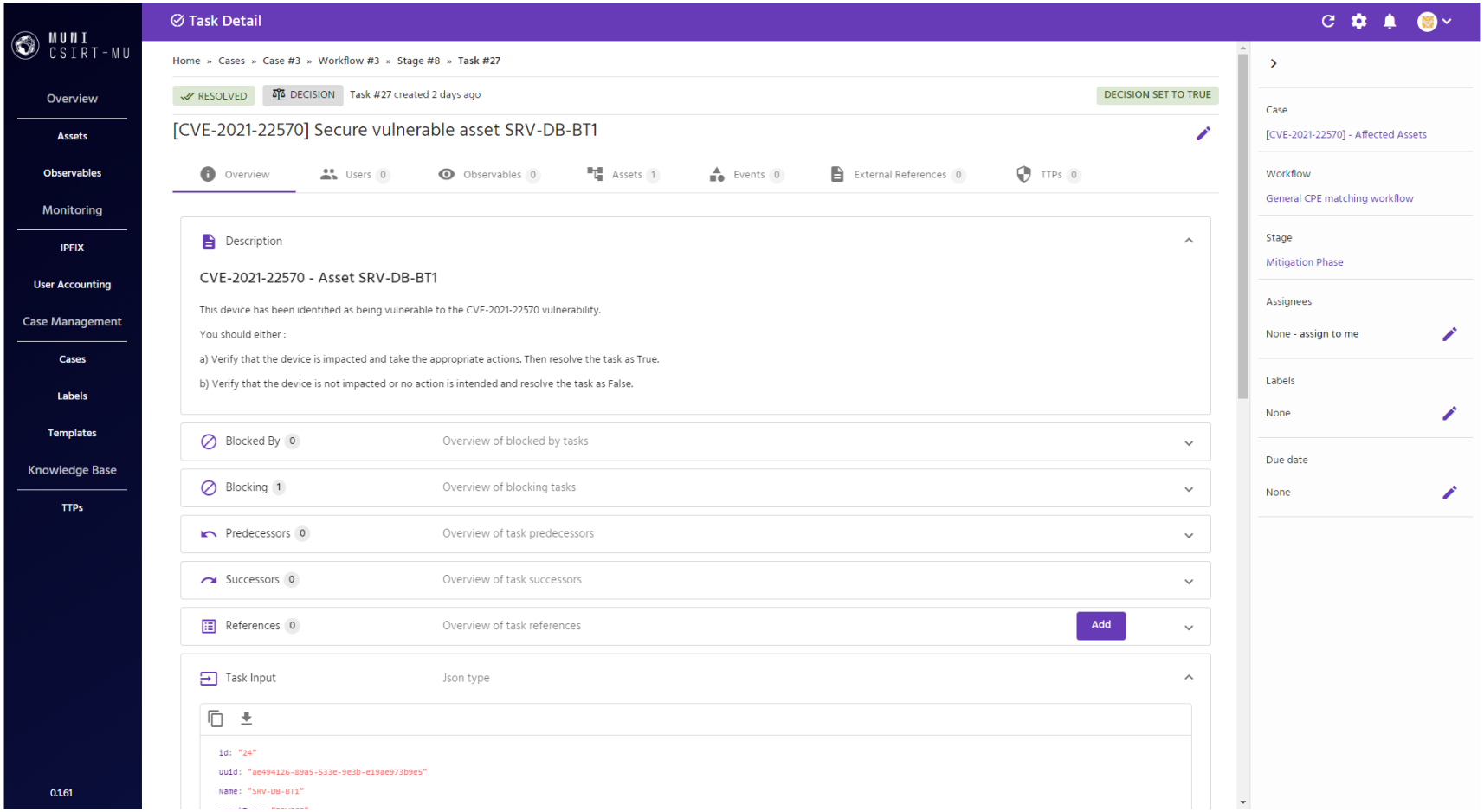
1. Operátor zadá CVE číslo do formuláře webové aplikace komponenty „CVE uploader“ z Výsledku č. 2 [1].
2. Je spuštěn pod-proces, který v databázi aktiv vyhledá ta aktiva, pro která je zadaná zranitelnost relevantní.
3. Pod-proces vytvoří případ a jeho pracovní postup, fáze pracovního procesu a úkoly, dle předpisu pracovního postupu pro provedení CPE párování.
4. Řešitel případu otevírá analytický pohled případu a postupuje dle popisku případu, pracovního postupu, jednotlivých fází pracovního postupu a úkolů.
5. Řešitel případu vidí, že dle odpovídající šablony byl vytvořen pracovní postup párování CPE. Naviguje tedy na analytický pohled pracovního postupu.
6. První fází pracovního postupu je analýza, jejíž cílem by mělo být zjistit podrobnosti o dané zranitelnosti, její závažnosti, a rozhodnout, zda ji mitigovat. Řešitel případu naviguje na analytický pohled první fáze pracovního postupu.
7. Fáze analýzy obsahuje úkol s příznakem rozhodnutí. Řešitel přistupuje na analytický pohled úkolu. Řešitel musí rozhodnout, zda je potřeba zranitelnost na příslušných aktivech mitigovat. S rozhodováním mu pomůže popis CVE a hodnocení závažnosti zranitelnosti dle různých metrik poskytovaných NVD, či OpenCVE. V případě, že se řešitel rozhodne zranitelnost mitigovat, nastaví rozhodnutí na hodnotu pravda, čímž úkol uzavírá.
8. Po uzavření případu dojde k aktivaci fáze mitigace, jejímž hlavním úkolem je agregační úkol, který sdružuje jeho jednotlivé pod-úkoly. Počet pod-úkolů odpovídá počtu aktiv s detekovanou potenciální zranitelností. Řešitel v právě uzavřeném úkolu vidí, že byl otevřen následující agregační úkol, do kterého naviguje.
9. Rozhodovací pod-úkoly jsou automaticky vytvořeny příslušným pod-procesem. Do agregačního úkolu je možné manuálně přidat další pod-úkol mitigace zranitelnosti aktiva, která nebyla identifikována nástrojem CPE párování.
10. Řešitel postupně kontroluje aktiva provázané s jednotlivými pod-úkoly. Příslušné aktivum si řešitel může zobrazit v přepínací kartě „Assets“. Úkol je uzavřen rozhodnutím pravda v případě, že příslušné aktivum je zasaženo danou zranitelností a byly provedeny kroky k nápravě této zranitelnosti. Pokud aktivum není zasaženo, uzavře řešitel úkol rozhodnutím nepravda.
11. Předešlý krok je opakován pro všechny pod-úkoly.
12. Ve chvíli, kdy jsou všechny pod-úkoly vyřešené, může řešitel vyřešit agregační úkol kliknutím na tlačítko „Join outputs of blocking tasks“. Dojde ke spojení výstupů jednotlivých pod-úkolům. V12.1.ýstupem agregačního úkolu je tedy seznam aktiv, jejichž zranitelnost byla mitigována. Úkol a fáze mitigace je uzavřena.
13. Následuje závěrečná fáze pracovního postupu s názvem revize. Fáze obsahuje úkol přípravy zprávy o příslušném CVE. Tento úkol má příznak automatizovaný a automatická spoušť. To znamená, že jeho řešení a uzavření je řízeno automatizovaným pod-procesem. Ke spuštění automatizovaného pod-procesu dojde automaticky, okamžitě po otevření úkolu. Řešitel případu může zkontrolovat, zda je výstup úkolu nastaven správně a úkol uzavřen.
14. Všechny fáze pracovního postupu jsou úspěšně uzavřeny, případ neobsahuje žádné další pracovní postupy. Řešitel případu naviguje na analytický pohled případu a uzavírá jej.



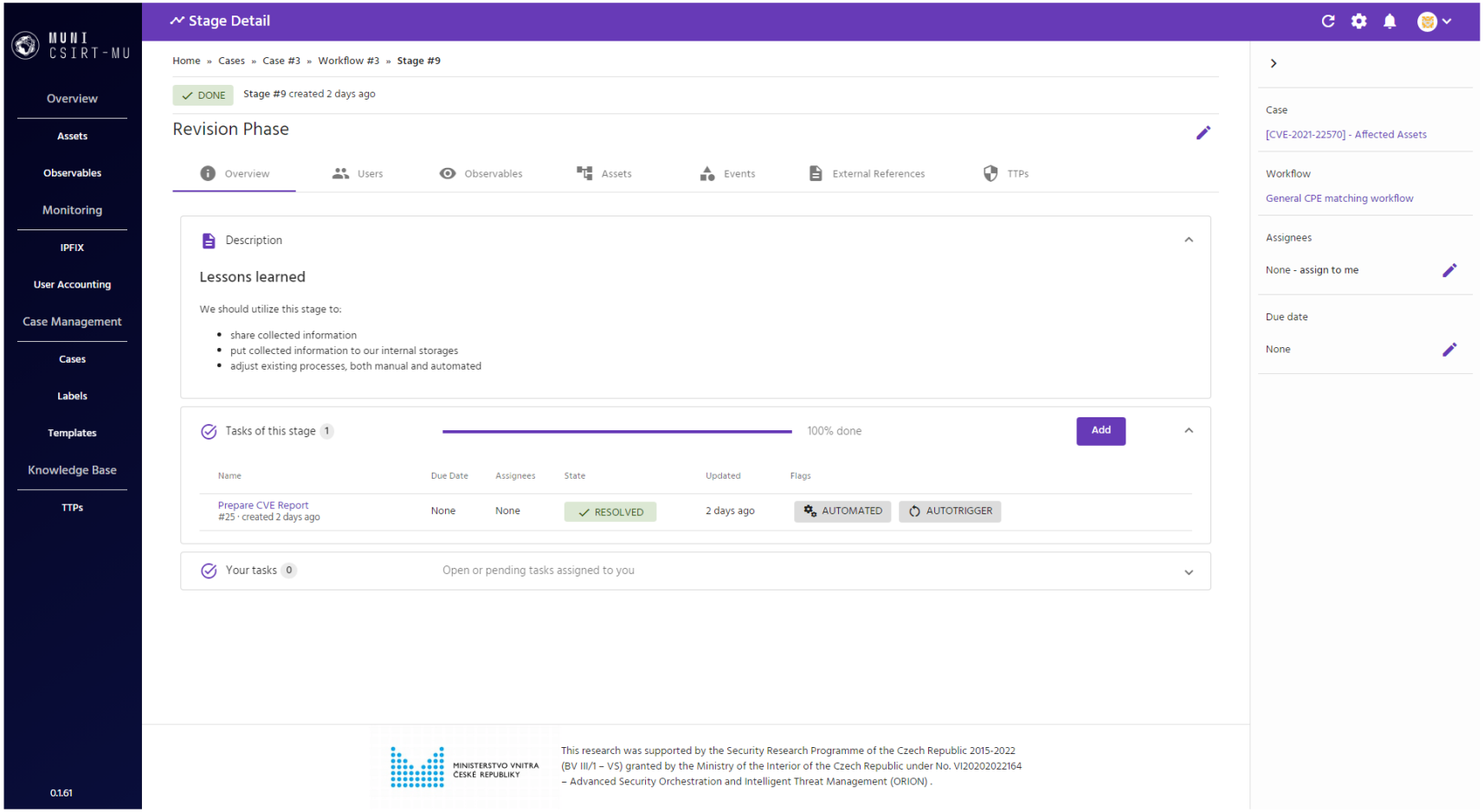

*Obrázek 33. Rozhodovací úkol analýzy detailů CVE z kroku 7*




*Obrázek 34. Agregační úkol mitigace zranitelností aktiv z kroku 9*




*Obrázek 35. Rozhodovací úkol mitigace zranitelnosti na konkrétním aktivu z kroku 10*



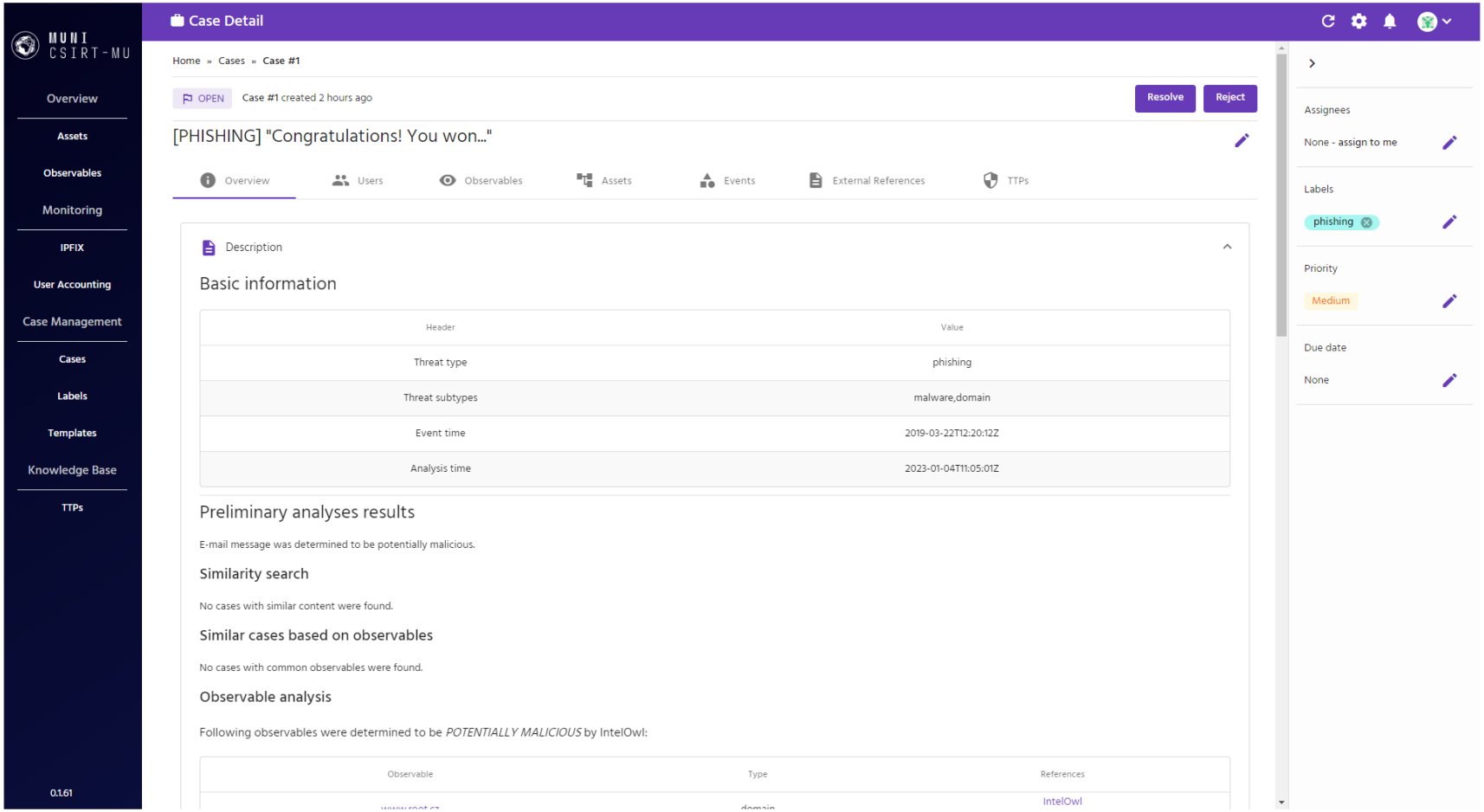

*Obrázek 36. Závěrečná fáze pracovního postupu revize z kroku 13*

## Demonstrace řešení případu pro hrozby typu Phishing

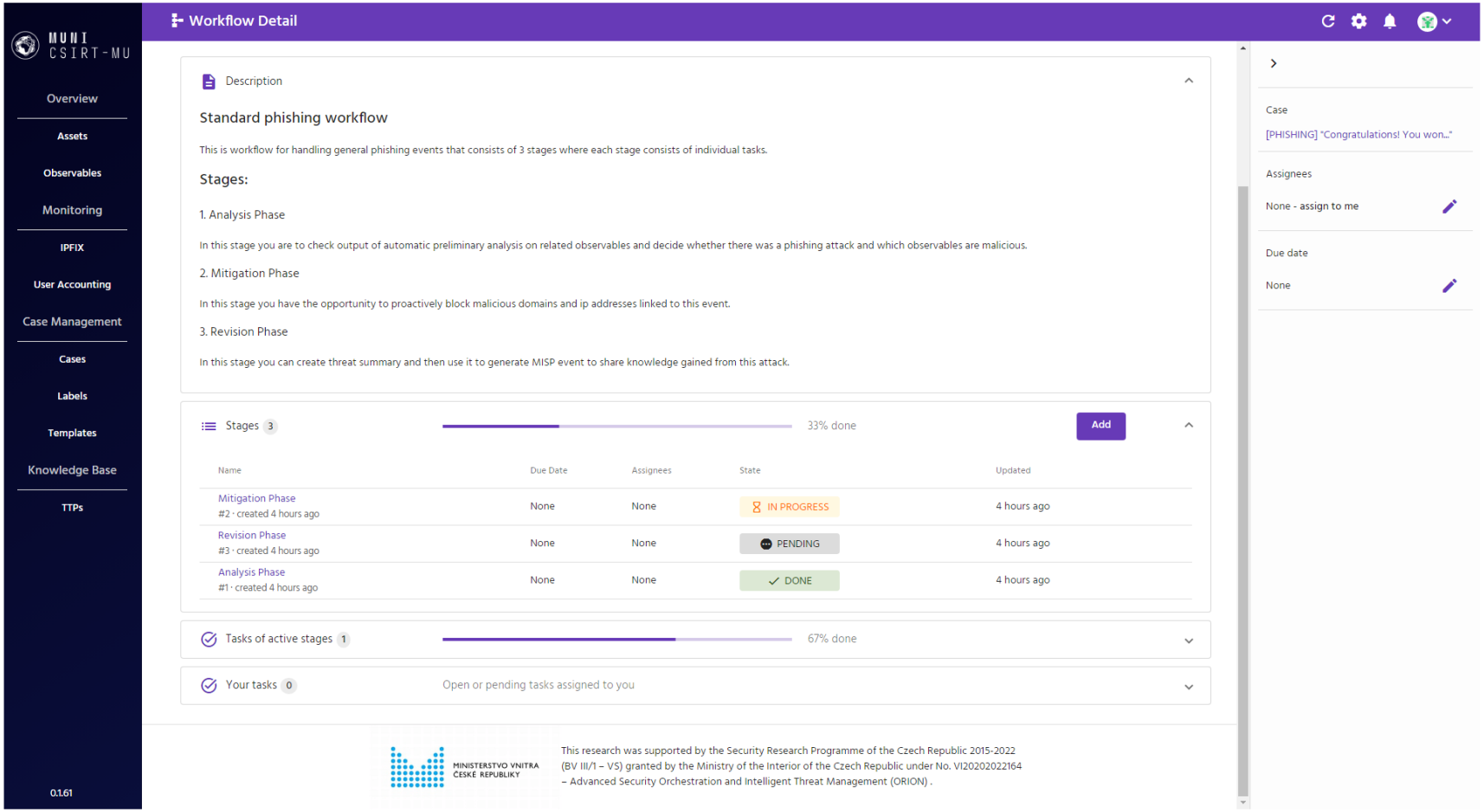
Případ hrozby typu „Phishing“ je praktickým využitím „Orchestračního scénaře pro hrozby typu „Phishing“ z Výsledku č. 2 [1]. Následuje popis jednotlivých kroků řešení případu v kontextu „Uživatelského rozhraní“.

1. Operátor, nebo uživatel nahraje podezřelý soubor do formuláře webové aplikace „Phishing Uploader“ z Výsledku č. 2 [1].
2. Je spuštěn pod-proces, který nahraný soubor analyzuje. Pod-proces hledá například podobnost se známými podvodnými texty, škodlivé domény, škodlivé soubory. Detailní popis je ve Výsledku č. 2 [1].
3. Na závěr pod-proces vytvoří nový případ, jeho pracovní postup, fáze pracovního postupu a odpovídající úkoly, dle předpisu pracovního postupu pro phishing.
4. Řešitel případu otevírá analytický pohled případu a postupuje dle popisku případu, pracovního postupu, jednotlivých fází pracovního postupu a úkolů.
5. V popisku případu se řešitel základní informace, zejména o jaký typ hrozby se jedná, zda existuje podobnost s jinými případy na základě obsahu phishingu, nebo zájmových údajů. Dále popisek obsahuje seznam analýz zájmových údajů a souborů, včetně odkazů do konkrétních analytických nástrojů.
6. Řešitel případu vidí, že dle odpovídající šablony byl vytvořen pracovní postup obecného phishingu. Naviguje tedy na analytický pohled pracovního postupu.
7. Pracovní postup je složen z fází analýza, mitigace a revize. První fází je analýza. Řešitel naviguje na její analytický pohled.
8. Fáze analýza začíná úkolem s příznakem rozhodnutí, jehož cílem je na základě informací z analytických nástrojů rozhodnout o tom, zda se jedná o phishing a zda je potřeba v řešení případu pokračovat.
9. Pokud řešitel phishing potvrdí, je automaticky spuštěn automatizovaný úkol, jehož cílem je uložení a zachování dat o phishingu pro účely podobnostního vyhledávání v budoucích případech phishingu. Dále je spuštěno několik agregačních úkolů, jejichž cílem je vybrat z informací o phishingu všechny škodlivé domény, IP adresy a soubory. Pokud automatizovaný pod-proces nezachytil nějakou z podezřelých domén, IP adres, nebo souborů, může řešitel vytvořit úkol manuálně a přidat ho tak do agregace.
10. Pro každý zájmový údaj (domény, IP adresy a soubory) je vytvořen samostatný rozhodovací úkol. Tyto úkoly blokují příslušný nadřazený agregační úkol. Řešitel postupně prochází rozhodovací úkoly k jednotlivým zájmovým údajům. V případě úkolů vztahujícím se k IP adresám, má řešitel k dispozici vybrané síťové toky, kde daná IP adresa vystupuje jako příjemce, či odesílatel. K těmto úkolům jsou vytvořeny vazby na uživatele, kteří s IP adresou komunikovali. Úkoly, jejichž cílem je analyzovat jednotlivé přílohy phishingu obsahují odkaz na soubor v zabezpečeném úložišti S3 a odkazy na analýzy automatizovaných nástrojů. Posledním typem jsou úkoly, jejichž cílem je analýza jednotlivých domén, které se v phishingovém emailu vyskytly. Tyto úkoly obsahují základní informace o doméně, jako je její přesměrovací URL, snímek obrazovky webové stránky a DNS záznamy. Na základě DNS záznamů může proces na pozadí vytvořit další rozhodovací úkoly pro IP adresy, vytvořené úkoly bude s kořenovým doménovým úkolem spojovat reference.
11. Ve chvíli, kdy jsou všechny pod-úkoly vyřešené, může řešitel vyřešit agregační úkol kliknutím na tlačítko „Join outputs of blocking tasks“. Dojde ke spojení výstupů jednotlivých pod-úkolů, výstupem agregačního úkolu je tedy seznam zájmových údajů daného typu (IP adres, domén, nebo souborů), které byly vybrány a označeny jako škodlivé.
12. Následujícím úkolem je manuálně projít vybrané emailové adresy a vybrat ty, které by měly být zablokované. K výběru může řešitel využít specializovanou grafickou komponentu k výběru objektů ve formátu JSON.

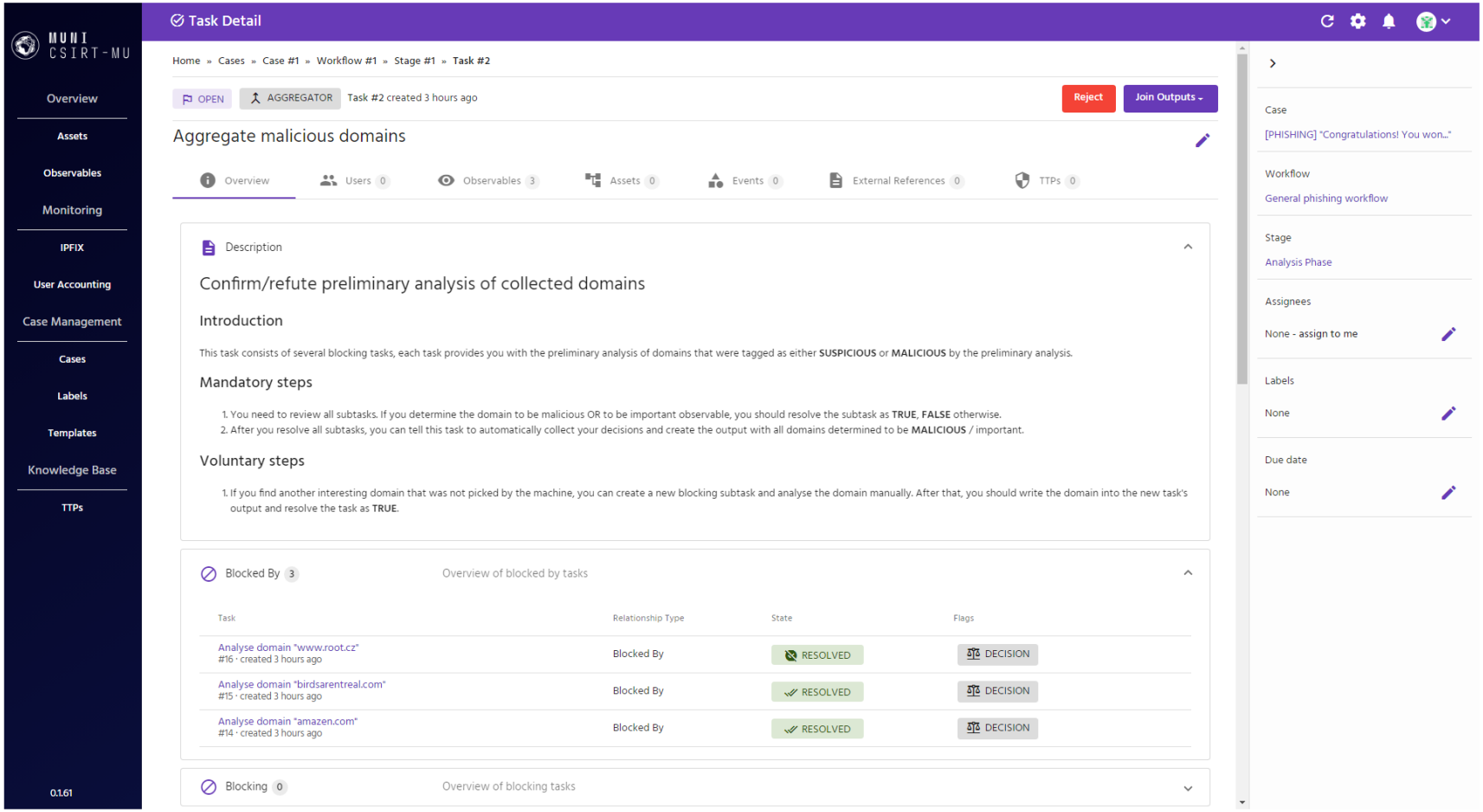
13. Po agregaci výstupů a uzavření všech úkolů je uzavřena analytická fáze případu a otevírá se mitigační fáze. Tato fáze sestává pouze z automaticky spuštěných plně automatizovaných úkolů, které zablokují všechny IP adresy, emailové adresy a domény, které byly v předchozí fázi označeny jako škodlivé.
14. Poté co automatizované pod-procesy provedou blokaci, zapíší výstupy jednotlivých úkolů a úkoly uzavřou, končí mitigační fáze pracovního postupu a začíná poslední, revizní fáze. V analytickém pohledu této fáze je řešitel pověřen vypracováním souhrnné zprávy případu, případně zlepšit existující procesy na základě ponaučení z aktuálního případu.
15. Prvním úkolem této fáze je vytvoření souhrnné zprávy. Tato zpráva je vytvářena automaticky spuštěným automatizovaným pod-procesem. Po dokončení pod-procesu je souhrn zapsán jako výstup úkolu a úkol uzavřen.
16. Posledním úkolem je vytvořit odpovídající událost v systému MISP. Řešitel projde souhrn případu, který byl vytvořen v předchozím úkolu. Po zkontrolování korektnosti souhrnu spustí automatizovaný pod-proces kliknutím na tlačítko „Trigger“. Pod-proces na základě souhrnu vytvoří událost v systému MISP a na výstup úkolu je zapsán odkaz na tuto událost.
17. Všechny fáze pracovního postupu jsou úspěšně uzavřeny, případ neobsahuje žádné další pracovní postupy. Řešitel případu naviguje na analytický pohled případu a uzavírá jej.



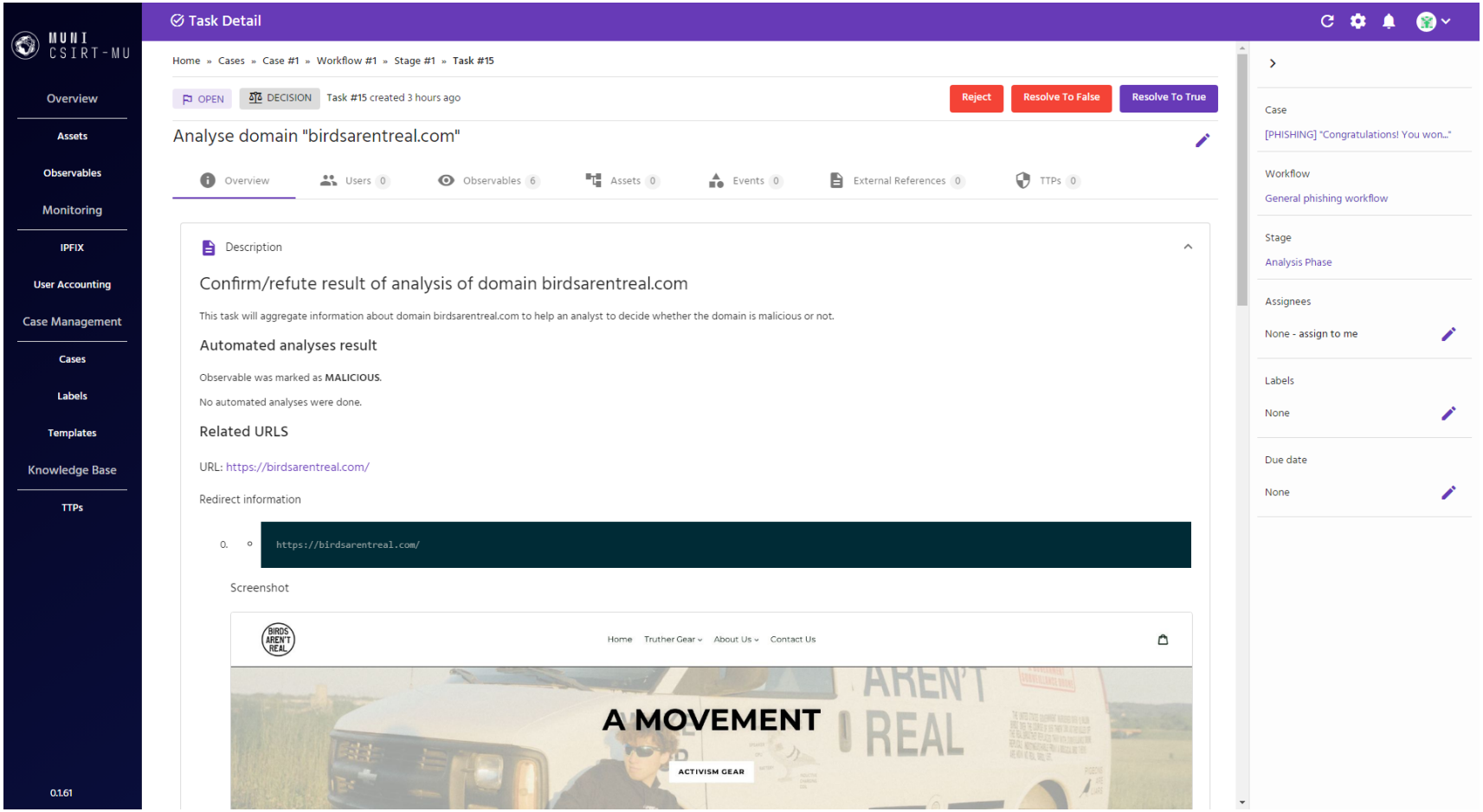

*Obrázek 37. Analytický pohled případu pro hrozby typu Phishing z kroku 5*




*Obrázek 38. Analytický pohled výchozího pracovního postupu pro hrozby typu Phishing z kroku 7*




*Obrázek 39. Agregační úkol mitigace škodlivých domén z kroku 9*




*Obrázek 40. Rozhodovací úkol potvrzující blokaci konkrétní domény z kroku 10*

# 5. Přílohy

## 5.1. Reference


[1] D. Tovarňák a ř. tým, „VI20202022164 - Pokročilá orchestrace bezpečnosti a inteligentní řízení hrozeb (ORION) - Výsledek č. 2 (R2),“ Masarykova univerzita, Brno, 2022c.

[2] K. D. Swenson, Mastering the Unpredictable, Meghan-Kiffer Press, 2010.

[3] D. Tovarňák a ř. tým, „VI20202022164 - Pokročilá orchestrace bezpečnosti a inteligentní řízení hrozeb (ORION) - Výsledek č. 1 (R1),“ Masarykova univerzita, Brno, 2022.

[4] D. Tovarňák, S. Špaček and J. Vykopal, "Traffic and log data captured during a cyber defense exercise," *Data in Brief,* Srpen 2020.

[5] D. Tovarňák a ř. tým, „VI20202022164 - Pokročilá orchestrace bezpečnosti a inteligentní řízení hrozeb (ORION) - Výsledek č. 3 (R3),“ Masarykova univerzita, Brno, 2022.

## 5.2. Poděkování

Tento výzkum byl podpořen Programem bezpečnostního výzkumu České republiky v letech 2015–2022 (BV III/1-VS), který je poskytovaný Ministerstvem vnitra ČR, v projektu č. VI20202022164 Pokročilá orchestrace bezpečnosti a inteligentní řízení hrozeb.