Universidade Federal do Rio de Janeiro

Construção de um sistema de monitoração de servidores inspirado em internet das coisas

Fernando Sampaio Pereira dos Anjos e Pedro Santos Eusébio

2020

Universidade Federal
do Rio de Janeiro
Escola Politécnica

Construção de um sistema de monitoração de servidores inspirado em internet das coisas

Fernando Sampaio Pereira dos Anjos e Pedro Santos Eusébio

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica,Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Prof. Guilherme Horta Travassos D.Sc

Rio de Janeiro

Dezembro de 2020

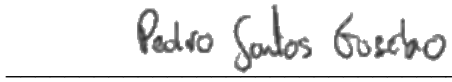
Construção de um sistema de monitoração de servidores inspirado em internet das coisas

Fernando Sampaio Pereira dos Anjos e Pedro Santos Eusébio

PROJETO DE GRADUAÇÃO SUBMETIDA AO CORPO DOCENTE DO CURSO DE ENGENHARIA DO COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO

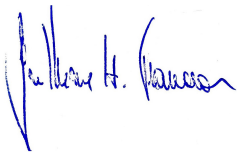
Autores:

Fernando Sampaio Pereira dos Anjos



Pedro Santos Eusébio

Orientador:



Prof. Guilherme Horta Travassos, D.Sc.

Examinador:

Claudio Miceli de Farias, D.Sc.

Examinador:

Diego Leonel Cadette Dutra, D.Sc.

Rio de Janeiro

Dezembro/2020

Anjos, Fernando Sampaio Pereira dos

Eusébio, Pedro Santos

Construção de um Sistema de Monitoramento de Servidores Inspirado em Internet das Coisas. – Rio de Janeiro: UFRJ / Escola Politécnica, 2020.

X, 65 p.: il.; 29,7 cm.

Orientador: Guilherme Horta Travassos

Projeto de Graduação – UFRJ / Escola Politécnica / Curso de Engenharia de Computação e Informação, 2020.

Referências Bibliográficas: p. 64-65.

1. Introdução 2. A escolha da metodologia Lean Inception 3. Definindo os requisitos através da Lean Inception 4. Construindo o MVP 5.Organização do código e exemplos. I. Travassos, Guilherme Horta. II. Universidade Federal do

Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Computação e Informação. III. Título.

Agradecimentos

Agradecemos a todos os responsáveis por nossa formação acadêmica. Professores, pais, amigos e cidadãos brasileiros, que possibilitam a existência da universidade pública. Obrigado, professor Guilherme Horta Travassos D.Sc., pela orientação e dedicação em nos desafiar e nos tornar melhores engenheiros.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

Construção de um Sistema de Monitoramento de Servidores Inspirado em Internet das Coisas

Fernando Sampaio Pereira dos Anjos e Pedro Santos Eusébio

Dezembro/2020

Orientador: Guilherme Horta Travassos

Curso: Engenharia de Computação e Informação

Com o intuito aplicar os conhecimentos adquiridos de engenharia de software e de criar uma aplicação-base capaz de ser evoluída para uso profissional dos autores, este trabalho implementa um painel de controle (*dashboard*) capaz de gerenciar dados de diferentes servidores a partir de um protocolo de comunicação pré-definido. A ideia central é poder monitorar, de forma centralizada e organizada, a saúde de diferentes dispositivos a partir de dados por eles disponibilizados. Três questões fundamentais nortearam a escolha de tecnologias de desenvolvimento: reutilização de componentes; usabilidade do painel de controle; e baixa latência na atualização dos dados monitorados.

Palavras-chave: painel de controle; sistema de monitoração; servidores internet das coisas.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Engineer.

Building a Server Monitoring System Inspired on Internet of Things

Fernando Sampaio Pereira dos Anjos and Pedro Santos Eusébio

December/2020

Advisor: Guilherme Horta Travassos

Course: Computer Engineering

To apply the acquired knowledge of software engineering and create a base application capable of being evolved for the authors' professional use, this work implements a dashboard capable of managing data from different servers using a predefined protocol. In a centralized and organized way, the main idea is to monitor the health of various devices based on data made available by them. Three fundamental concepts guided the choice of development technologies: component reuse, the usability of the control panel, and low latency in updating monitored data.

Keywords: control panel; dashboard; monitoring system; servers; internet of things.

Sumário

Capítulo 1 – Introdução

Antes de abordarmos a construção da solução, apresentaremos a base para o entendimento das nossas motivações e decisões.

1.1. Contextualização e Motivação

Vivemos em uma época onde tudo ao nosso redor se conecta. Num futuro não tão distante, todos os nossos dispositivos serão capazes de se comunicar através da internet. Estamos presenciando a era da "Internet das coisas" [1].

Imagine a situação em que você possui diversas "coisas" com capacidade de conexão com a internet e, assim, permitir que você acompanhe seu funcionamento e possa interagir (sua geladeira; seu celular; um servidor de um projeto pessoal; uma coleira inteligente para seu cão, dentre outros), seria ótimo uma forma de monitorar a saúde dos recursos desses dispositivos num único local. Nesse intuito nasceu a ideia para o nosso projeto. Queremos construir um painel de controle (*dashboard*) capaz de coletar dados de diversos dispositivos, aqui chamados de "coisas".

A motivação principal para a construção desse *dashboard* nasceu do nosso dia a dia profissional. Parte das nossas tarefas consiste em receber pedidos de suporte de clientes cuja aplicação, por nós licenciada, está com algum tipo de problema sistêmico ou físico. Dessa forma, gostaríamos de ter para cada aplicação-servidor se comportando como uma "coisa" capaz de expor dados sobre sua saúde. Isso poderia nos proporcionar uma forma de monitoramento capaz de detectar problemas mais rapidamente ou, até mesmo, antecipar falhas iminentes.

Para tornar isso possível, foi necessário elaborar uma arquitetura capaz de alimentar o *dashboard* em tempo real e atualizar sua visualização à medida em que cada "coisa" disponibiliza seus dados. Também foi preciso definir um padrão de comunicação para a troca de dados, com isso, desenvolvedores de sistemas terceiros, por exemplo, poderiam chamar APIs (*application programming interface*) do *dashboard* para passar quaisquer tipos de dados de processos que desejam monitorar. Cabe ressaltar que o projeto foi inspirado em "internet das coisas", mas não implementa essa ideia de forma pura, pois os

dispositivos não disponibilizam seus dados em um middleware para que qualquer um os consuma, eles chamam API's do servidor do *dashboard* diretamente. Na verdade, podemos dizer que nosso projeto acaba se aproximando de "DevOps" [2], onde nosso principal objetivo é o monitoramento contínuo, sendo, então, um foco maior no "Ops" (Operacional) do que em "Internet das Coisas".

Em todas as etapas deste trabalho, mantivemos como foco aplicar diversos conceitos fundamentais aprendidos durante a graduação, tais como reutilização de *software*, orientação a objetos e modelagem de bancos de dados.

1.2. Objetivos

A proposta deste trabalho é projetar e implementar um painel de controle (*dashboard)* de monitoração da saúde de outros dispositivos espalhados fisicamente. Assim, em um único local, podemos ter uma visão geral sobre o status de dispositivos (se estão em funcionamento ou não); sobre o desempenho físico (uso de recursos como memória, CPU, temperatura); sobre métricas mais específicas informadas pelo dispositivo (quantidade de ovos na geladeira, por exemplo).

O objetivo deste trabalho é atender à necessidade específica do nosso dia a dia profissional, monitorando servidores/aplicações de nossos clientes, por isso, definimos uma forma de comunicação a ser seguida pelas coisas sem grandes compromissos com padrões externos. Nosso foco esteve em construir todo o ecossistema, distribuindo o esforço entre a definição da arquitetura e protocolo básico de comunicação; construção do *dashboard* aplicando as tecnologias escolhidas; aplicação da metodologia "*Lean Inception*" [3] para definição e evolução dos requisitos de *software*. Essa metodologia foi essencial para termos clareza acerca das funcionalidades que realmente eram necessárias para a existência do produto. Veremos todos os passos da *Lean Inception* que tornaram possível a tomada de decisões, como o "descarte de funcionalidades", por exemplo, sem maiores dificuldades. Podemos dizer que um diferencial do nosso trabalho no que tange a aplicação da "*Lean Inception",* em relação a outras análises [4] ou aplicações dela, foi o fato de que alguns dos passos da metodologia puderam ser descartados, uma vez que não se aplicavam à nossa realidade. Como exemplo, podemos citar o "orçamento", que, aqui,

não era uma restrição. Além disso, a nossa aplicação não está limitada ao monitoramento de servidores, na verdade, ela é pode monitorar qualquer dispositivo capaz de se comunicar na linguagem que definimos, isso significa que, no limite, o *dashboard* pode exibir e analisar dados de uma geladeira, de um carro, de um aplicativo terceiro qualquer ou, como é o foco aqui, do nosso próprio *software* e do servidor no qual ele está hospedado.

1.3. Definição do problema

No contexto profissional dos alunos envolvidos nesse trabalho, existe a necessidade de concentrar informações acerca de dispositivos (servidores) e de processos remotos (do sistema operacional e de aplicações específicas). Hoje o acompanhamento da saúde e a investigação de um problema é feita acessando as aplicações expostas via *http* por esses servidores ou através de conexão direta via RDP (*remote desktop protocol*).

O cenário descrito acima não é eficiente, pois primeiramente, requer algum tipo de aviso dos usuários impactados por problemas para que um membro da equipe de suporte faça algum tipo de intervenção. Esse tipo de dependência não apenas coloca os usuários em uma condição emocional delicada, na qual é necessário que “sintam uma dor para reportar”, como também deixam a equipe de suporte sem previsibilidade da necessidade de intervenções.

Assim sendo, o objetivo principal do projeto é inverter esse fluxo, fazendo que com que os usuários não mais tenham que reportar um problema, mas que o *dashboard* seja capaz de prevê-los antes que aconteça a partir de regras cadastradas e da disponibilização de métricas a partir das “coisas”. Para que o *dashboard* seja capaz de, constantemente, aumentar seu grau de cobertura, é fundamental que sua construção seja genérica em relação à sua modelagem e interface. Isso também se aplica ao protocolo de comunicação proposto.

1.4. Estrutura do documento

No capítulo 1, inserimos o leitor no contexto abordado, expusemos a motivação deste trabalho, os objetivos do mesmo, o problema que desejamos resolver e discriminamos a estrutura do texto.

No capítulo 2, apresentamos a metodologia escolhida para guiar nosso processo de desenvolvimento, já adiantando alguns pontos positivos do seu uso.

Focamos o capítulo 3 no trabalho de levantamento de requisitos, detalhando todos os passos da metodologia. Apresentamos os artefatos resultantes e como foi o cronograma de desenvolvimento do sistema.

No capítulo 4, apresentamos os detalhes técnicos da construção do mínimo produto viável, que referenciaremos pela sigla "MVP" (*minimum viable produc*t) [5]. Aproveitamos para ilustrar os desafios encontrados.

No capítulo 5, apresentamos a organização do código e alguns trechos que julgamos mais relevantes para ilustrar algumas decisões e o funcionamento do *backend* e do *frontend*.

No capítulo 6, apresentamos as telas do sistema e avaliamos os pontos positivos e possíveis melhorias do resultado obtido.

Por fim, no capítulo 7, apresentamos nossas conclusões e expectativas de evolução em trabalhos futuros, apresentando os motivos para a lista de evoluções proposta

Capítulo 2 – A escolha da metodologia Lean Inception

Aqui discutiremos o porquê desta escolha como metodologia para chegarmos no mínimo produto viável e no quê ela consiste. Além disso, aproveitamos para adiantar quais foram os nossos principais desafios e recompensas ao longo de sua execução.

2.1. O motivo para buscarmos uma metodologia

Em virtude da execução deste trabalho por mais de uma pessoa, surgiu a oportunidade de experimentarmos algo que não tivéssemos feito antes. Dentro do contexto atual de desenvolvimento de *software*, o termo "ágil" é comum, mas nem sempre isso significa "curta duração". No nosso dia-a-dia como desenvolvedores que trabalham lado-a-lado na mesma empresa, estamos acostumados a um processo mais livre, talvez, até enxuto demais. Cada um de nós é responsável por um módulo do sistema que desenvolvemos, de maneira bastante isolada, assim, quase não sentimos a necessidade de um processo mais longo e estruturado de "alinhamento de pessoas para a definição de um mínimo produto viável". Parte dessa sensação vem do fato de termos entrado na empresa com grande parte do produto já definido.

Comparando o que estamos acostumados com a nossa realidade neste projeto, percebemos que havia um grande desafio: construir algo novo, a partir "do nada". Por isso, as primeiras conversas rapidamente escalaram para um processo "caótico" e pouco conclusivo. Neste momento ficou claro que seria necessário um caminho bem estruturado para a definição dos requisitos do sistema.

A busca nos levou a descobrir a *Lean Inception* [3]. Parte deste conceito (*Inception*) não seria uma novidade, dado que já conhecíamos como a primeira fase do RUP (*Rational Unified Process*) [6]. O diferencial, que nos chamou a atenção, foi a adição da palavra "*Lean*", que indicava exatamente o que pudemos perceber ao estudar a metodologia: era uma versão "enxuta" que visa minimizar desperdício de tempo, esforços e recursos da equipe. Em relação ao objetivo, também se encaixava com as nossas necessidades: alinhar nosso entendimento e detalhes do mínimo produto viável que queríamos construir.

2.2. Sobre a metodologia

Sua ideia fundamental está pautada em um processo ágil e incremental de desenvolvimento de sistemas, visando um produto enxuto (com duração menor de construção) que representa um MVP. A aplicação da metodologia foi bastante exercitada em modelo de workshop pelo autor Paulo Caroli, resultando em um guia bastante detalhado em seu livro [3]. Como o próprio autor define, a metodologia consiste em um workshop colaborativo capaz de alinhar um grupo de pessoas acerca do produto mínimo viável a ser construído.

2.3. Desafios e recompensas

Como desenvolvedores acostumados a um alto volume de entregas na nossa rotina, sem dúvida, foi um desafio focar grande parte do nosso tempo em tarefas que não envolviam código. Apesar de ter sido um primeiro desafio, trouxe grandes recompensas para as etapas seguintes, pois vivenciamos um baixíssimo nível de retrabalho em função de definições equivocadas.

Outro ponto interessante a destacar foi a "mudança de ecossistema" para a construção dos artefatos que surgiriam com a aplicação da metodologia. No mundo atual, ninguém imagina uma ferramenta diferente do computador para armazenar e organizar suas ideias e conclusões. Como veremos, a *Lean Inception* propõe o uso de recursos físicos e dinâmicas presenciais que já não são comuns na rotina da maioria dos desenvolvedores. Entretanto, podemos afirmar que foi um diferencial e descoberta incríveis. O simples ato de ter uma "conexão física" com as ideias que estão sendo trabalhadas trouxe mais uma recompensa inesperada: o "prazer" na execução desta fase da construção do software, a sensação foi a de "jogar um jogo de tabuleiro" onde todos venceram ao final.

Capítulo 3 – Definindo os requisitos através da Lean Inception

Neste capítulo, mostraremos todos os passos da metodologia que nos levou à definição dos requisitos que compuseram os MVPs. Começamos com a apresentação de itens básicos usados ao longo de todo o workshop, para depois detalhar cada passo e artefato resultante.

3.1. Aspecto físico da metodologia

Como adiantamos no capítulo anterior, algumas particularidades sobre a metodologia nos surpreenderam e trouxeram grande produtividade para a execução dos passos. Por conta disso, inclusive, levamos alguns deles para dentro do nosso ambiente profissional.

3.1.1. Sala de guerra

Durante a execução da *Lean Inception*, mantivemos a sala das nossas casas reservadas para o projeto. Dessa forma, tínhamos um local dedicado para a concentração das conversas e cartolinas, post-its, quadro branco, café, dentre outros. Isso se mostrou bastante produtivo e nos fez perceber como a interação pessoal, ao invés de videoconferência, traz um fator humano interessante para o processo.

Figura 1. Sala de guerra

3.1.2. Post-its coloridos

O uso de artefatos físicos como *post-its* coloridos em cartolinas trouxe muito combustível para a nossa criatividade. Pode parecer um detalhe sem grande valor, mas o uso de recursos físicos para trabalhar nossas ideias se mostrou extremamente eficaz e divertido. Pudemos perceber como grandes projetos podem se beneficiar dessa dinâmica.

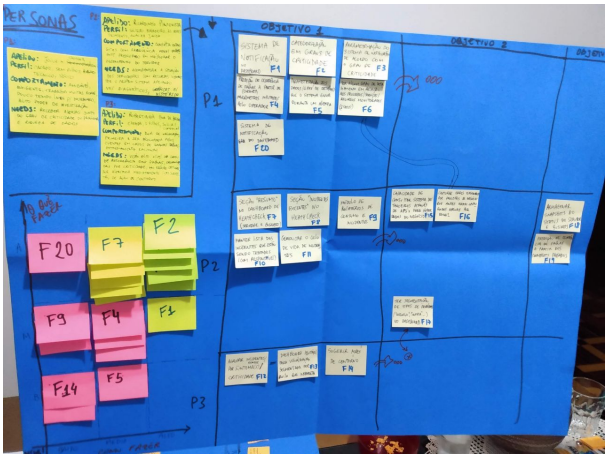


Figura 2. Exemplo de uso dos post-its coloridos no processo

3.1.3. Estacionamento de ideias

Uma forma de progredir e, ao mesmo tempo, não deixar completamente de lado algumas boas ideias, usamos esse recurso como forma de armazenamento. Assim, poderíamos revisitar alguns pontos em momento mais adequado para não poluir as discussões. Novamente, podemos ressaltar como um pequeno detalhe pode fazer grande diferença. A simples existência de uma área reservada para o descarte ou para deixar funcionalidades na “geladeira”, para avaliação posterior, criou uma tranquilidade que não poderíamos imaginar. Esse item nos ajudou bastante para seguirmos em frente sem o “medo” de estarmos deixando algo de fora, pois teríamos a “chance de revisitar o assunto mais tarde”.

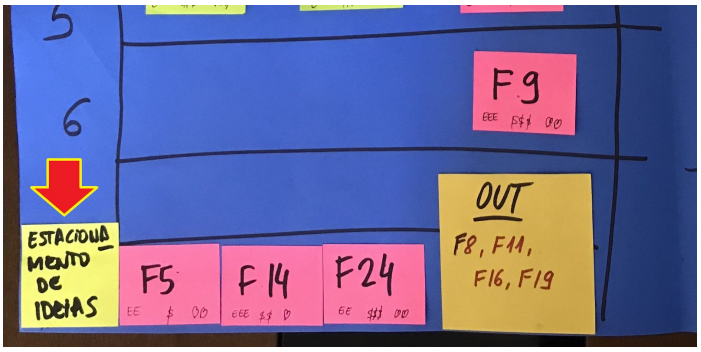


Figura 3. Estacionamento de ideias

3.2. Atividades e artefatos resultantes

Abaixo vamos destacar as principais atividades executadas e os artefatos por elas gerados.vamos destacar os principais abaixo:

3.2.1. Visão do produto / É; Não é; Faz; Não faz

Nestas atividades cada um escreveu frases curtas para adicionarmos ao nosso quadro que organiza os *post-its* do workshop, o qual chamaremos de *canvas*, e analisarmos os "*matchs*" entre elas. Na "visão" do produto, por exemplo, pudemos mapear semelhanças e combinações interessantes:

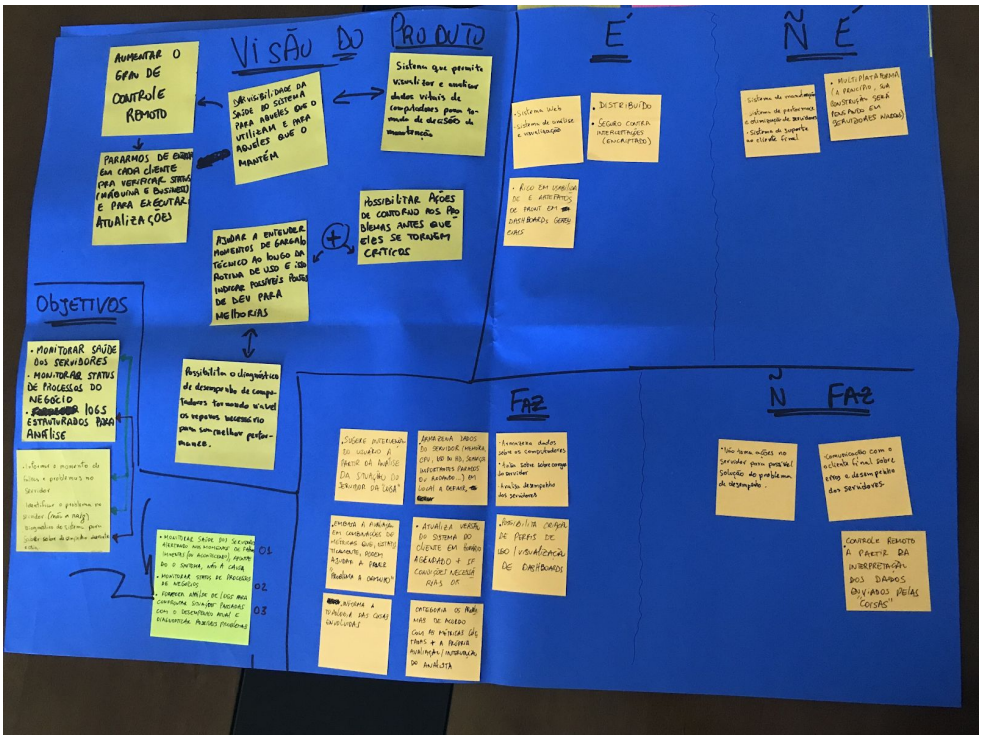


Figura 4. Visão do produto e suas características

Destacando os tópicos da figura acima, gerados no *brainstorm*, separados por cada um para mostrar o alinhamento de ideias acerca do projeto:

Visão:

- Fernando:
  - “Aumentar o grau de controle remoto”;
  - “Pararmos de entrar em cada cliente para verificar o status”;
  - “Dar visibilidade da saúde do sistema para os que o utilizam e para os que o mantém”;
  - “Ajudar a entender momentos de gargalo técnico ao longo da rotina de uso e isso indicar possíveis pontos de “dev” para melhorias”;
  - “Possibilitar ações de contorno aos problemas antes que eles se tornem críticos”.
- Pedro:

    - “Sistema que permite visualizar e analisar dados vitais de computadores para tomada de decisão da manutenção”;
    - “Possibilitar o diagnóstico de desempenho de computadores tornando viável os reparos necessários para melhor performance”.

É:

- Fernando:
    - “Distribuído”;
    - “Seguro contra interceptações (encriptado)”;
    - “Rico em usabilidade e artefatos de “front” em *dashboards* gerenciais”.
- Pedro:
    - “Sistema web”;
    - “Sistema de análise e visualização”.

Não É:

- Fernando:
    - “Multiplataforma (a princípio, será construído pensando em servidores windows)”.
- Pedro:
    - “Sistema de manutenção”;
    - “Sistema de performance e otimização de servidores”;
    - “Sistema de suporte ao cliente final”.

Faz:

- Fernando
    - “Sugere intervenção a partir da análise do servidor”;
    - “Armazena dados do servidor (memória, cpu, uso de hd, serviços importantes parados ou rodando…)”;
    - “Embasa a avaliação em combinações de métricas que, estatisticamente, podem ajudar a prever “problema a caminho”;

    - “Atualiza versão do sistema do cliente em horário agendado + IF condições necessárias”;
    - “Possibilita criação de perfis de uso/visualização de dashboards”;
    - “Informa a topologia das coisas evoluídas”;
    - “Categoriza os problemas de acordo com as métricas coletadas + a própria avaliação/intervenção do analista”;
- Pedro
    - “Armazena dados sobre os computadores”;
    - “Avisa sobre sobrecargas dos servidores”;
    - “Analisa desempenho dos servidores”;

Não Faz:

- Fernando
    - “Controle remoto a partir da interpretação dos dados enviados pelas “coisas””
- Pedro
    - “Não toma ações no servidor para possível solução do problema de desempenho”;
    - “Comunicação com o cliente final sobre erros e desempenho dos servidores.”

3.2.2. Objetivos

“Cada membro da equipe deve compartilhar o que entende como objetivo para o negócio, e os vários pontos de vista devem ser discutidos para chegar a um consenso sobre o que é realmente importante” [3]

- O1: “Monitorar a saúde dos servidores, alertando nos momentos de falhas iminentes (ou acontecendo), apontando o sintoma, não a causa”;
- O2: “Monitorar status de processos de negócios”;
- O3: “Fornecer análises de logs para confrontar situações passadas com o desempenho atual e diagnosticar possíveis problemas”.

3.2.3. Personas

Nesta atividade, desenhamos diferentes perfis de usuários do *dashboard* para entender, em passos seguintes, que funcionalidades serão necessárias. Escolhemos definir três “personas” buscando certa semelhança com alguns dos perfis presentes no nosso ambiente profissional. Apesar de podermos enxergar mais possibilidades do que “apenas três ‘personas’”, escolhemos as que pareciam ter maior probabilidade de uso diário do produto.

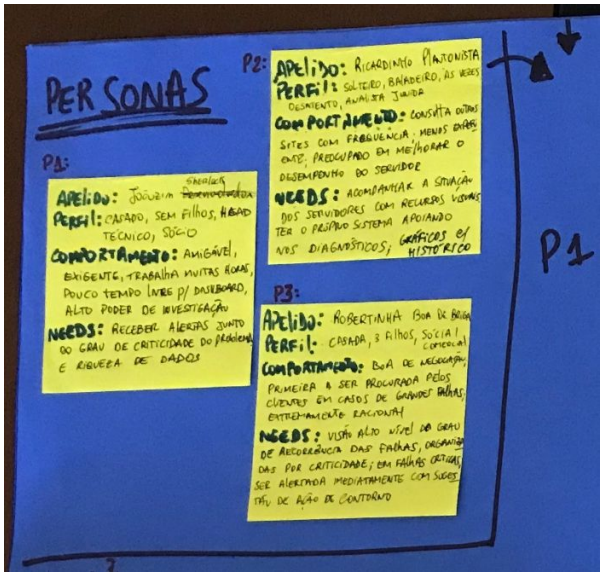


Figura 5. Descrição das “personas”

Abaixo, as personas que foram definidas:

- P1
    - Apelido: Joãozinho Sherlock
    - Perfil: casado; sem filhos; head técnico; sócio
    - Comportamento: amigável; exigente; trabalha muitas horas; pouco tempo livre para acessar o *dashboard*; alta capacidade de investigação
    - Necessidades: Receber alertas com grau de criticidade do problema e riqueza de dados para análise
- P2:
    - Apelido: Ricardinho Plantonista

    - Perfil: solteiro; baladeiro; às vezes desatento
    - Comportamento: consulta outros sites com frequência; menos experiente; preocupado em melhorar o desempenho dos sistemas monitorados
    - Necessidades: acompanhar a situação dos servidores com recursos visuais; ter o próprio sistema apoiando nos diagnósticos; gráficos com histórico
- P3:
    - Apelido: Robertina Boa de Briga
    - Perfil: casada; 3 filhos; sócia; *head* comercial
    - Comportamento: boa de negociação; primeira a ser procurada pelos clientes em casos de grandes falhas; extremamente racional
    - Necessidades: visão alto nível do grau de recorrência das falhas, organizadas por grau de criticidade; em falhas críticas, ser avisada imediatamente, com sugestão de contorno

É importante ressaltar que a própria ordem de enunciação dos tópicos das personas auxiliou na definição das "necessidades" de cada uma. Afinal, um comportamento X acaba despertando uma necessidade Y de forma a corrigir uma deficiência da persona ou para atender a uma necessidade dela.

3.2.4. Funcionalidades

Nesta atividade, uma tabela com Personas nas linhas e Objetivos nas colunas é construída. Assim, ficou fácil enxergar que funcionalidades seriam necessárias. Nosso trabalho foi apenas escrever, em alto nível, que funcionalidades precisaríamos construir para que os objetivos fossem atendidos, levando em conta as características de cada "persona".

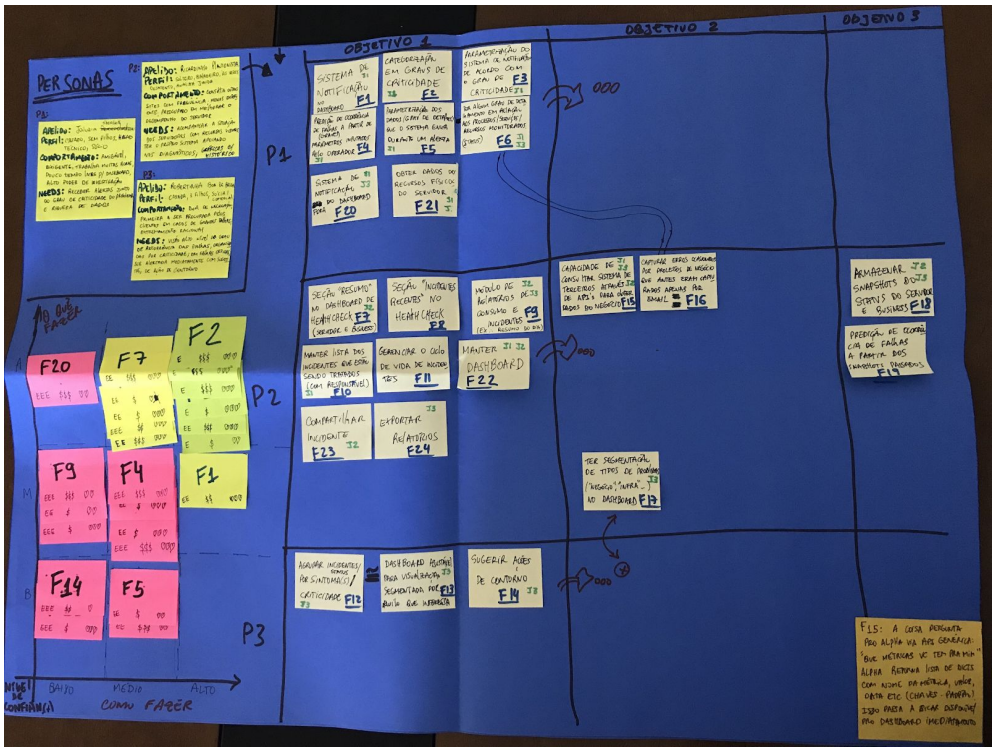


Figura 6. Personas; Objetivos e Funcionalidades; Mapa de semáforo

Abaixo, as funcionalidades identificadas nessa fase do projeto:

- F1: Sistema de notificação no *dashboard*
- F2: Categorização em graus de criticidade
- F3: Parametrização do sistema de notificação de acordo com o grau de criticidade
- F4: Predição de ocorrência de falhas
- F5: Parametrização do grau de detalhes enviados pelo sistema durante alertas
- F6: Detalhamento em relação aos processos/serviços/recursos monitorados
- F7: Seção “Resumo” no *dashboard* de *healthcheck*
- F8: Seção “Incidentes Recentes” no *healthcheck*
- F9: Módulos de relatórios de consumo e de incidentes
- F10: Manter lista de incidentes que estão sendo tratados (com responsável)
- F11: Gerenciar o ciclo de vida de incidentes

- F12: Agrupar incidentes por status/sintomas/grau de criticidade
- F13: *Dashboard* ajustável para visualização segmentada por aquilo que interessa
- F14: Sugerir ações de contorno
- F15: Capacidade de consultar sistemas de terceiros através de APIs para obter dados específicos (dados business tratado pelo sistema terceiro, por exemplo)
- F16: Capturar erros ocasionados por processos de negócio que antes eram capturados apenas por email
- F17: Ter segmentação por tipos de problema (“negócio”, “infra”...) no *dashboard*
- F18: Armazenar snapshots do status das métricas coletadas
- F19: Predição de ocorrência de falhas a partir dos snapshots passados
- F20: Sistema de notificações externas ao *dashboard*
- F21: Obter dados dos recursos físicos do servidor
- F22: Manter *dashboard*
- F23: Compartilhar incidente
- F24: Exportar relatórios

3.2.5. Jornadas

Nesta atividade, nosso objetivo era traçar a rotina de cada persona, isso nos ajudaria a entender duas coisas:

1. Se as funcionalidades listadas seriam suficientes para cobrir as jornadas das personas
2. Se algumas das funcionalidades poderiam ser descartadas, ou seja, se não eram fundamentais para o MVP que atende às jornadas

Para alcançar o objetivo, escrevemos qual funcionalidade seria necessária dentro do post-it de cada passo da jornada.

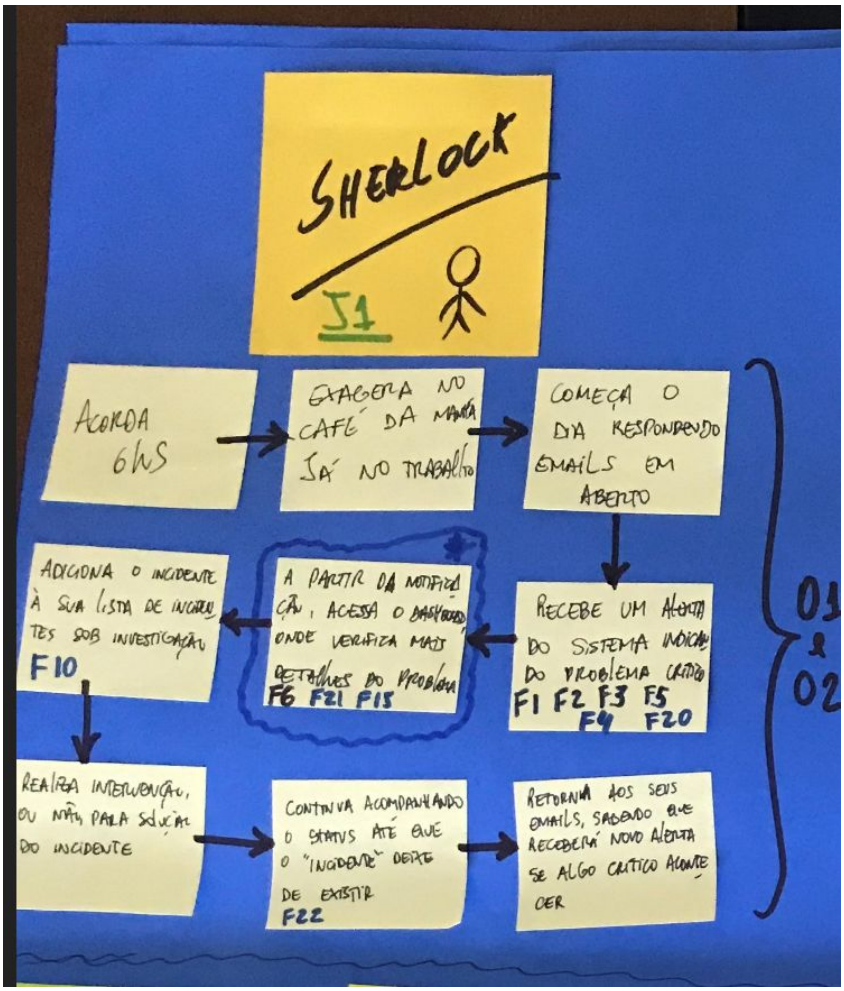


Figura 7. Jornada da persona 1 (Joãozinho Sherlock)

Jornada 1 (Persona 1): Acorda às 6hs > Exagera no café da manhã, já no trabalho > Começa o dia respondendo emails em aberto > Recebe um alerta do sistema indicando um problema crítico (F1, F2, F3, F4, F5, F20) > A partir da notificação, acessa o *dashboard*, onde verifica mais detalhes do problema (F6, F15, F21) > Adiciona o incidente à sua lista de incidentes sob verificação (F10) > Realiza intervenção, ou não, para solução do incidente > Continua acompanhando o status até que o incidente deixe de existir (F22) > Retorna aos seus emails, sabendo que receberá novo alerta se algo crítico acontecer

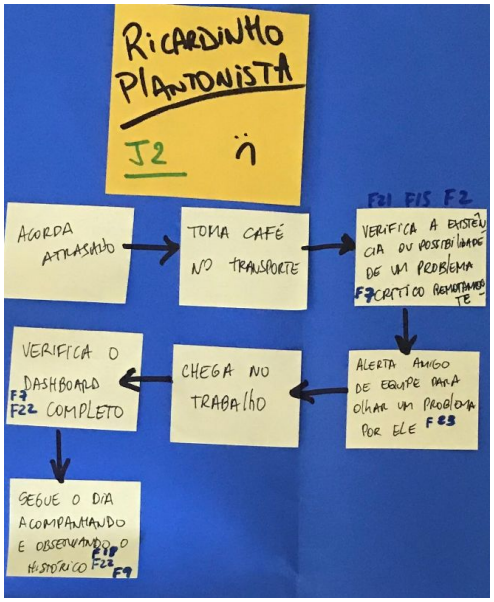


Figura 8. Jornada da persona 2

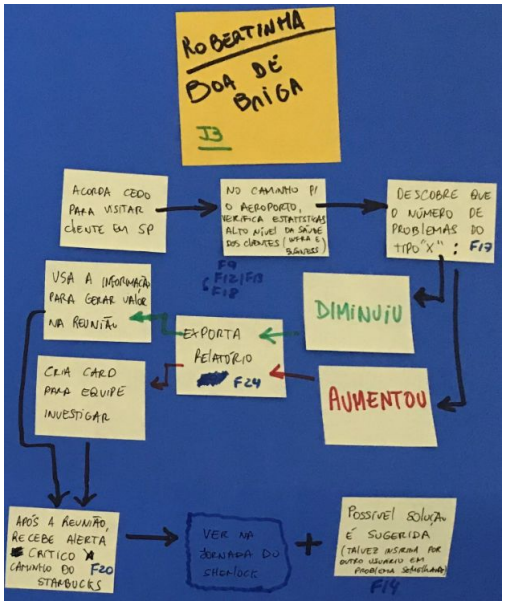


Figura 9. Jornada da persona 3

Jornada 2 (Persona 2): Acorda atrasado > Toma café no transporte > Remotamente, já verifica a existência ou possibilidade de um problema crítico (F2, F7, F15, F21) > Alerta amigo da equipe para verificar um problema enquanto não chega no trabalho (F23) > Chega no trabalho > Verifica o status completo de todas as coisas monitoradas (F7, F22) > Segue o dia acompanhando e observando o histórico para entender mais sobre a rotina dos sistemas monitorados (F9, F18, F22)

Jornada 3 (Persona 3): Acorda cedo para visitar cliente em SP > No caminho para o aeroporto, verifica "estatísticas alto nível" da saúde dos sistemas dos clientes (dados de infra e business) (F9, F12/13, F22) > Descobre que o número de problemas de um dado tipo aumentou ou diminuiu > Exporta relatórios (F24) > A partir da redução ou aumento de problemas, usa a informação para gerar valor para a reunião ou para criar tarefa interna investigar > Após a reunião, recebe alerta crítico à caminho do Starbucks (F20) > Segue passos da jornada 1, ou não > Possível solução é sugerida (inserida por outro usuário em incidente passado semelhante) (F14)

3.2.6. Descartando funcionalidades

Aqui podemos destacar uma grande recompensa do processo. Sabemos quão difícil é descartar ideias em qualquer processo criativo. O desenvolvimento de *software*, portanto, compartilha desta dificuldade. No entanto, a simples execução dos passos anteriores nos trouxe clareza sobre o quê poderia ser descartado ou, simplesmente, postergado para um momento futuro. Isso foi alcançado após cruzarmos os passos das jornadas com as funcionalidades (anotando neles o número da funcionalidade que atendia ao passo em questão).

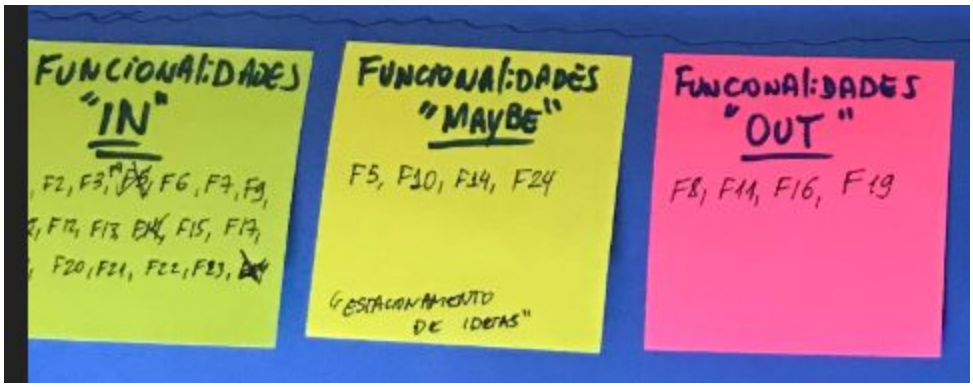


Figura 10. Funcionalidades essenciais para as jornadas e descartáveis

Assim, descartamos:

- F8: Seção “Incidentes Recentes” no healthcheck
- F11: Gerenciar o ciclo de vida de incidentes
- F16: Capturar erros ocasionados por processos de negócio que antes eram capturados apenas por email
- F19: Predição de ocorrência de falhas a partir dos snapshots passados

Gráfico do semáforo

Esta atividade também foi bastante surpreendente e trouxe clareza para o processo de codificação que viria a seguir. Até aqui, todas as funcionalidades estavam descritas em post-its amarelos. Agora cada um ganhou uma cor de acordo com a sua posição no gráfico. Vejamos abaixo a forma de classificação:

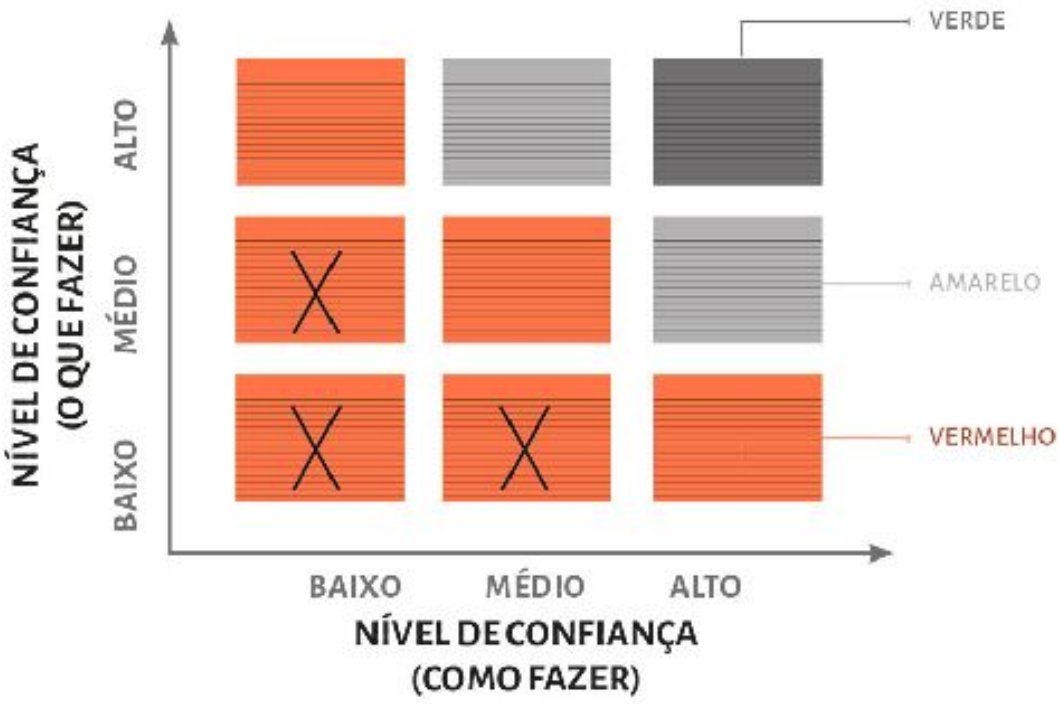


Figura 11. Gráfico de semáforo CAROLI [3]

Seguimos a orientação do autor [3]: "Classificamos cada funcionalidade combinando o nível de confiança técnico (como fazer) e o nível de confiança de UX (*user experience*) e do negócio (o que fazer). Dessa maneira, cada funcionalidade recebe uma cor relativa ao nível de confiança. Se uma funcionalidade ficar na parte inferior esquerda do gráfico (marcada com um "X"), considere descartá-la ou gaste mais tempo para esclarecê-la".

A partir desse exercício, tivemos uma primeira projeção de esforço em relação a cada uma das funcionalidades, o que foi fundamental para organização do cronograma de desenvolvimento, como veremos.

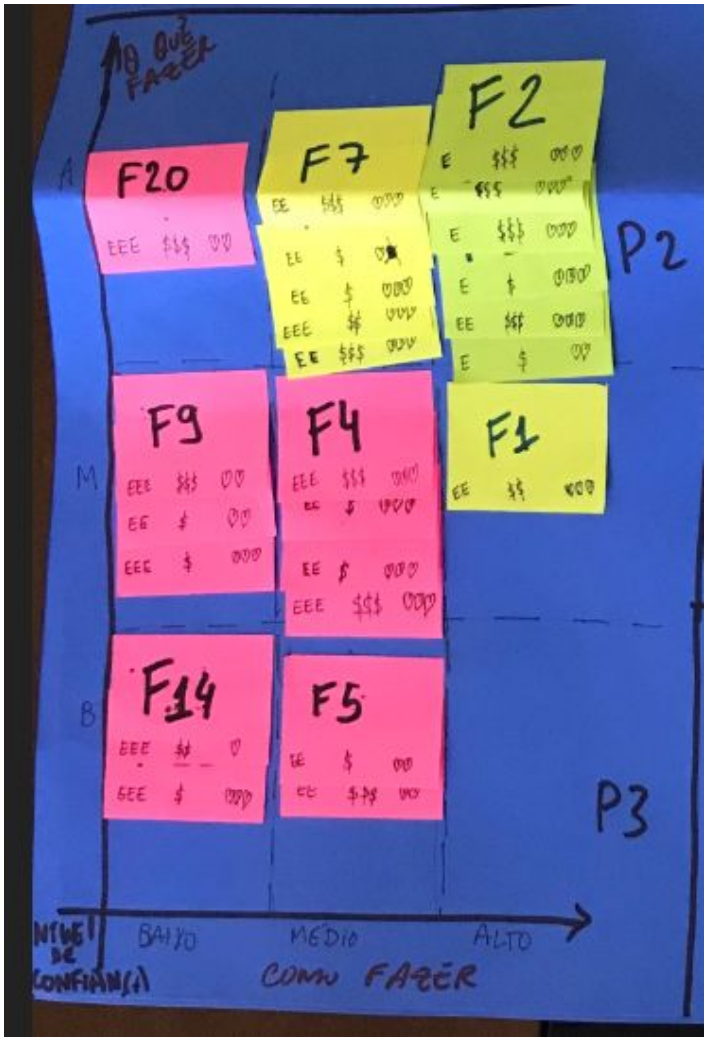


Figura 12. Funcionalidades no gráfico de semáforo

Revisão Técnica, de Negócio e de UX

Esse foi outro ponto importante para entendermos que funcionalidades poderiam ser descartadas (ou postergadas), além de contribuir com novas avaliações que serão usadas para organizar o cronograma de desenvolvimento. A atividade consistia classificar cada funcionalidade, em uma escala de um a três, nos seguintes quesitos:

- Esforço (“E”): qual a expectativa de esforço para desenvolver a funcionalidade

- Negócio (“$”): quanto valor econômico a funcionalidade agrega para o produto
- UX (“❤”): quanto para a experiência do usuário a funcionalidade agrega

É claro que estávamos sujeitos a avaliações “equivocadas” por inexperiência na execução do *workshop*, mas, mesmo assim, percebemos que o simples exercício de classificação já foi capaz de apontar possíveis problemas com funcionalidades que nem eram tão essenciais.

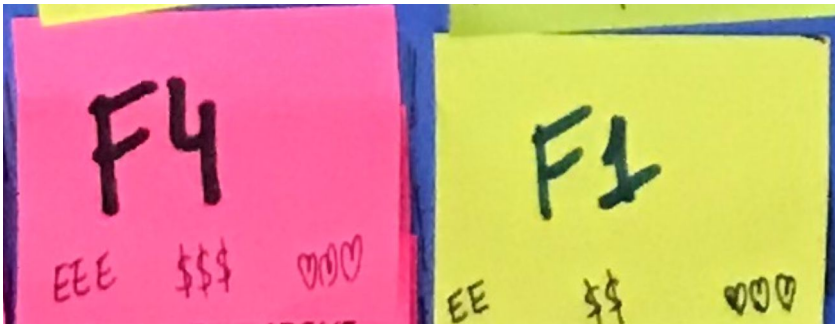


Figura 13. Classificação de funcionalidades

Definindo as ondas de desenvolvimento e os MVPs

Uma onda de desenvolvimento pode ser entendida com a união de diferentes funcionalidades em um espaço de tempo para a construção. No nosso caso, cada onda teve duração aproximada de uma semana.

A sensação que tivemos foi de que tudo que foi feito até aqui tinha como objetivo tornar a definição dessas ondas a menos “dolorosa” possível. Isso porque foi absolutamente trivial usar os artefatos anteriores, seguir algumas pequenas regras e sair com a sequência de desenvolvimento bem estruturada. As regras eram:

1. Uma onda pode conter, no máximo, três cartões (funcionalidades)
2. Uma onda não pode conter mais de um cartão vermelho
3. Uma onda não pode conter apenas cartões vermelhos ou amarelos
4. A soma de esforço (“E”) de uma onda não pode passar de cinco

5. As quantidades de “$” e de “❤” não podem ser menores que quatro por onda
6. Se um cartão depende de outro, então ele deve estar em uma onda anterior

Como o próprio autor define [3]: “A regra 1 limita o número de funcionalidades que estão sendo trabalhados ao mesmo tempo. Isso evita o acúmulo de itens de trabalho parcialmente completos, aumentando o foco para as poucas funcionalidades priorizadas por onda. As regras 2, 3 e 4 evitam um período de trabalho desequilibrado, com muita incerteza ou muito esforço. A regra 5 garante o foco constante na entrega de alto valor para o negócio e para os usuários. A regra 6 evita problemas de dependência entre funcionalidades”

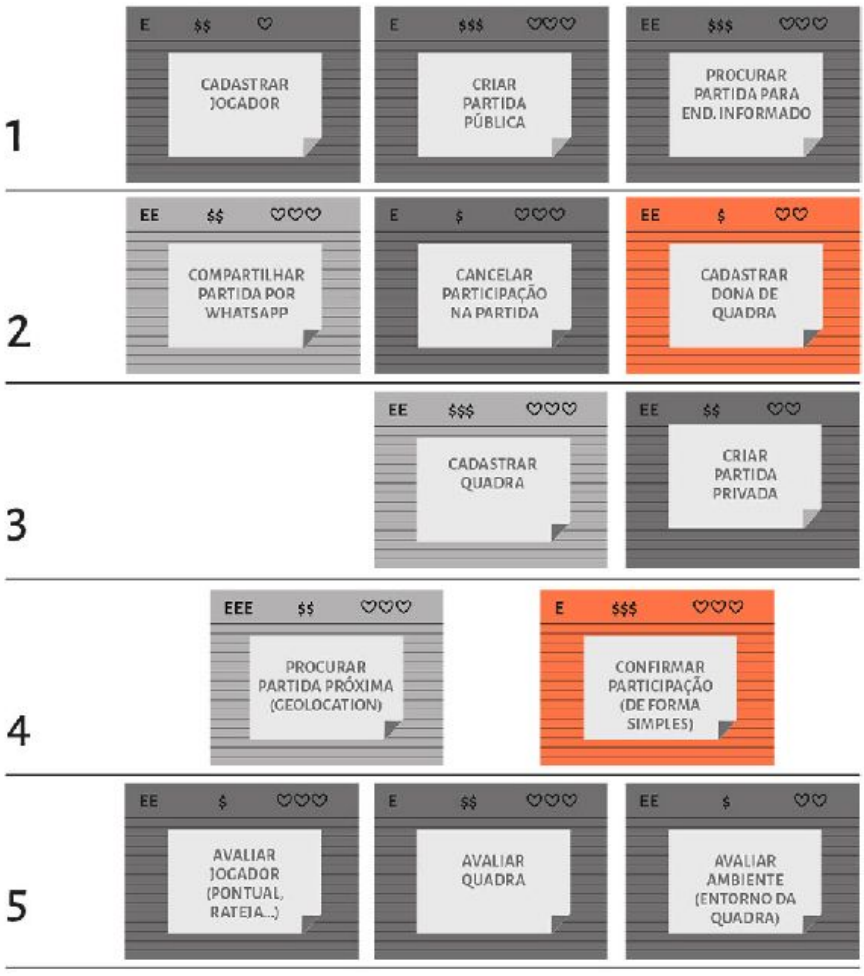


Figura 14. Divisão em ondas de desenvolvimentos [3]

Para definir as nossas ondas, além de seguirmos as regras listadas acima, também re-visitamos o diagrama com as jornadas dos usuários, pois isso nos dava indicadores de uma ordem cronológica entre as funcionalidades. Após essa análise, pudemos, finalmente, agrupar em ondas de desenvolvimento de forma e definir MVPs:

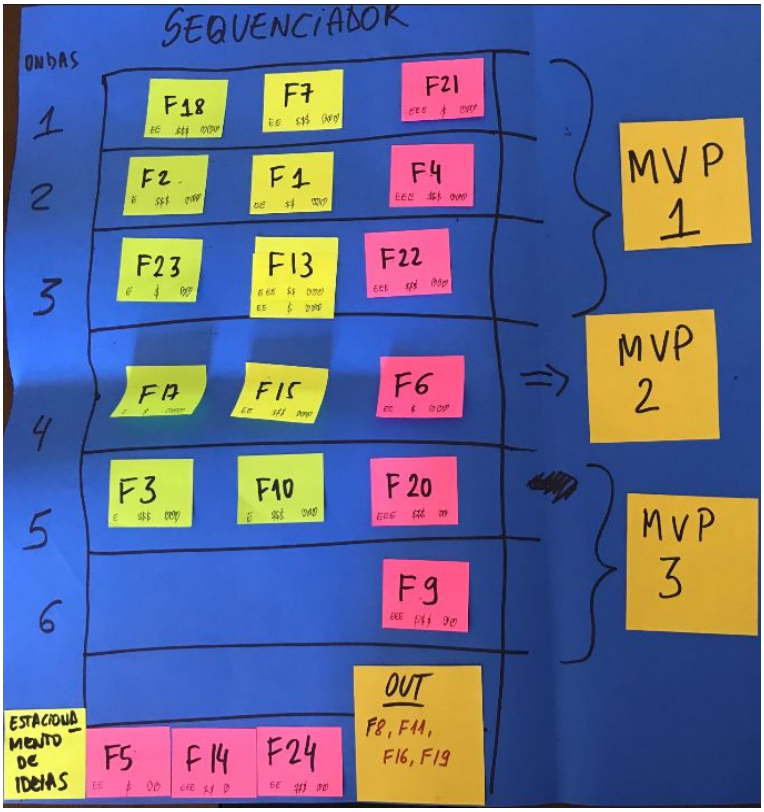


Figura 15. Ondas de desenvolvimento do projeto

É importante ressaltarmos que as regras 4 e 5 foram feridas, e que não era possível desviar desse resultado. Isso nos deu um indicador de que nossa avaliação (“E”, “$” e “❤”) estava descalibrada em relação ao ideal para a metodologia. Certamente algo que o autor [2] coordena entre os participantes do workshop para obter resultados mais adequados. Abaixo, as funcionalidades de cada MVP foram:

- MVP 1:
    - Onda de desenvolvimento 1:
        - F18: Armazenar snapshots do status das métricas coletadas

        - F7: Seção “Resumo” no *dashboard* de *healthcheck*
        - F21: Obter dados dos recursos físicos do servidor
    - Onda de desenvolvimento 2:
        - F2: Categorização em graus de criticidade
        - F1: Sistema de notificação no dashboard
        - F4: Predição de ocorrência de falhas
    - Onda de desenvolvimento 3:
        - F23: Compartilhar incidente
        - F13: *Dashboard* ajustável para visualização segmentada por aquilo que interessa + F12: Agrupar incidentes por status/sintomas/grau de criticidade (funcionalidades foram agrupadas em um único item)
        - F22: Manter *dashboard*

- MVP 2:
    - Onda de desenvolvimento 4:
        - F17: Ter segmentação por tipos de problema (“negócio”, “infra”...) no *dashboard*
        - F15: Capacidade de consultar sistemas de terceiros através de APIs para obter dados específicos (dados business tratado pelo sistema terceiro, por exemplo)
        - F6: Detalhamento em relação aos processos/serviços/recursos monitorados
- MVP 3:
    - Onda de desenvolvimento 5:
        - F3: Parametrização do sistema de notificação de acordo com o grau de criticidade
        - F10: Manter lista de incidentes que estão sendo tratados (com responsável)
        - F20: Sistema de notificações externas ao dashboard

- Onda de desenvolvimento 6:
    - F9: Módulos de relatórios de consumo e de incidentes

Capítulo 4 – Construindo o MVP

Neste capítulo serão detalhadas as tecnologias e ferramentas utilizadas; a modelagem de dados; e a arquitetura interna da solução proposta. Além disso, citaremos os principais desafios técnicos enfrentados e as recompensas que pudemos sentir após superá-los.

4.1. Tecnologia utilizada

O *dashboard* foi desenvolvido em Python 3.7 [7] por sua flexibilidade e por nossa familiaridade com a linguagem. O uso do framework Django [8] de desenvolvimento web para Python trouxe grande velocidade ao projeto Como sistema gerenciador de banco de dados, usamos o PostgreSQL [9]. Para construção do front-end, utilizamos React [10] e ag-Grid [11].

4.2. Modelo de Dados

Nesta seção descrevemos as entidades de nosso modelo de dados detalhamos seus campos e os relacionamentos entre as entidades.

Thing

Representa o servidor ou qualquer outro dispositivo que fornecerá dados para o dashboard.

Atributos:

- Name: Nome da coisa para identificação visual no *dashboard*;
- Authentication Code: Código de identificação que vai ser usado para validar a autenticidade da coisa;
- Is Active: Booleano que informa se a coisa está ativa no sistema e deve ser autenticada/monitorada;
- Snapshot Interval: informa o período em que as medidas da coisa serão enviadas ao servidor. Essa propriedade será fundamental para definir se uma "coisa" está com problemas de comunicação ou não.

Thing Type

Classifica a Thing. O objetivo é criar uma identificação visual mais rápida para a coisa dentro do dashboard (exemplo: "servidor", "eletrodoméstico", "dispositivo móvel"...)

Atributos:

- Name: Nome do tipo.
- Icon: Ícone que será usado na exibição resumida da coisa.

Metric

Representa os metadados de alguma métrica informada pela Coisa.

Atributos:

- Identifier: identificador técnico da métrica (exemplo: “memory_use”)
- Name: Nome da métrica visível para os usuários (exemplo: “Consumo de memória”)
- Short Name: Nome curto para exibição resumida (em colunas de tabelas, por exemplo)
- Description: Uma texto explicando a métrica, o que ela representa e como foi calculada.
- Type: Representa o tipo da métrica. Pode ser: Booleano ou numérica.
- Measure Unit: Representa a unidade que serão enviados as medidas (exemplos: “metros”, “%”, “Mb”...)
- Critical Curve: Informa se o crescimento da medida é proporcional ou inversamente proporcional ao crescimento da criticidade. No MVP 1 pode ser: 'linear crescent' or 'linear decrescent'. Assim, teremos um indicador para interpretar se uma medida maior é algo “melhor ou pior” para a saúde do dispositivo monitorado

Metric Detail

Determina a relação entre as métricas e a coisa. É através da “metric detail” que sabemos se a coisa possui ou não aquela métrica no seu “catálogo de métricas monitoradas”, se ela é observada e se possui algum valor máximo.

Atributos:

- Max Value: Valor máximo da métrica para aquela coisa.
- Is Observed: Informa se essa métrica será observada para uma determinada coisa.

Snapshot

Representa uma fotografia da situação atual da Thing. Contém o conjunto de dados que foram enviados em uma determinada hora e a avaliação desse conjunto de medidas.

Atributos:

- Generated On: A data e horário em que os dados foram enviados.
- Worst metric status: O pior status de avaliação do conjunto de medidas. Pode ser: '*yellow*', '*red*' ou '*green*'. Tendo alguma medida avaliada como

'yellow', o status da Snapshot será *'yellow'*. Caso tenha alguma medida *'red'* será avaliado como *'red'*. Caso não tenha nenhuma medida *'yellow'* ou *'red'*, será classificado com *'green'*.

Measure

Representa o dado medido e avaliado de uma Thing para uma Metric. Essa métrica tem relação com um Snapshot e Metric.

Atributos:

- Current Value: Representa o valor medido.
- Evaluation Status: O status após a avaliação da medida. Pode ser: *'red'*, *'yellow'* ou *'Green'*.
- Extra_values: Um campo json para armazenarmos possíveis informações extras que julguemos interessantes, mas que não sejam fortes o suficiente para se tornarem colunas da tabela

Metric Evaluation Rule

Regra que será usada na avaliação de uma métrica específica para avaliar se o status da Measure.

Atributos:

- Name: nome para identificação visual da regra.
- Error Threshold: O valor usado para avaliar se a medida é *'red'*.
- Warning Threshold: O valor usado para avaliar se a medida é *'yellow'*.
- Threshold Type: Determina se a regra avaliará em valor absoluto ou em percentual.

Disaster Prediction Rule

Regra que será usada na avaliação de algum “desastre” da coisa. Essa regra é uma combinação das avaliações das medidas de um snapshot. Caso uma dessas combinações seja identificada, um desastre aconteceu. Tem uma relação com as coisas, de forma a determinar se essa coisa deve ser avaliada por essa regra ou não. Caso não tenha nenhuma coisa definida, deve ser usada para avaliar todas as coisas do sistema.

Atributos:

- Name: Nome para identificação da regra.
- Is Active: Informa se a regra está ativa ou não.

Disaster Requirement

Uma regra de predição de de desastre (DisasterPredictionRule) é composta por alguns requisitos, que chamamos de DisasterRequirement. Este, por sua vez, se relaciona com uma métrica e possui um atributo de threshold.

Na prática, seria como dizer que, por exemplo, um desastre existe (Disaster) se um servidor (Thing) possui a métrica "Uso de memória" (Metric) como "*warning*" (evaluation_status de Measure, que é avaliado de acordo com o minimal_threshold de DisasterRequirement) e, ao mesmo tempo, possui uso de CPU acima de 97%.

Atributos:

- Minimal Threshold: Exigência mínima que será usada para comparar com o Evaluation Status da medida (Measure). Pode ser: '*yellow*' (*warning*) ou '*Red*' (*error*).

Critical Time Period

Representa o período (intervalo) em que as regras de predição de desastre devem ser avaliadas. Caso a regra de predição de desastre não tenha nenhum período crítico, ela deve ser avaliada a todo instante (a cada novo dado da coisa). Assim, podemos definir, por exemplo, que o uso intenso de memória é crítico se acontecer entre uma certa faixa de horário e não em outra.

Atributos:

- Start Time: Início do período em que a regra deve ser avaliada.
- End Time: Final do período em que a regra deve ser avaliada.

Disaster

Classe que representa a ocorrência de um desastre de acordo com alguma regra cadastrada. Ele possui uma relação com o Snapshot e com a regra de predição de desastres.

Atributos:

- Description: Um texto gerado pelo sistema informando quais foram as medidas que foram infligidas, informando o Evaluation Status atual e o que não deveria passar.
- Solution: Um texto por extenso informando qual foi a solução usada para corrigir esse desastre. Posteriormente, será usado como sugestão para outros desastres que ocorram com a mesma regra e Thing.

Notification

Notificação gerada a partir de um desastre. Informa uma descrição do desastre e o meio em que foi enviado. Será usado para recuperar a informação do desastre e de forma que seja fácil e intuitivo o compartilhamento dessa informação.

Atributos:

- Description: Texto informando sobre o desastre.
- Channel: Informando qual canal foi enviado a notificação.

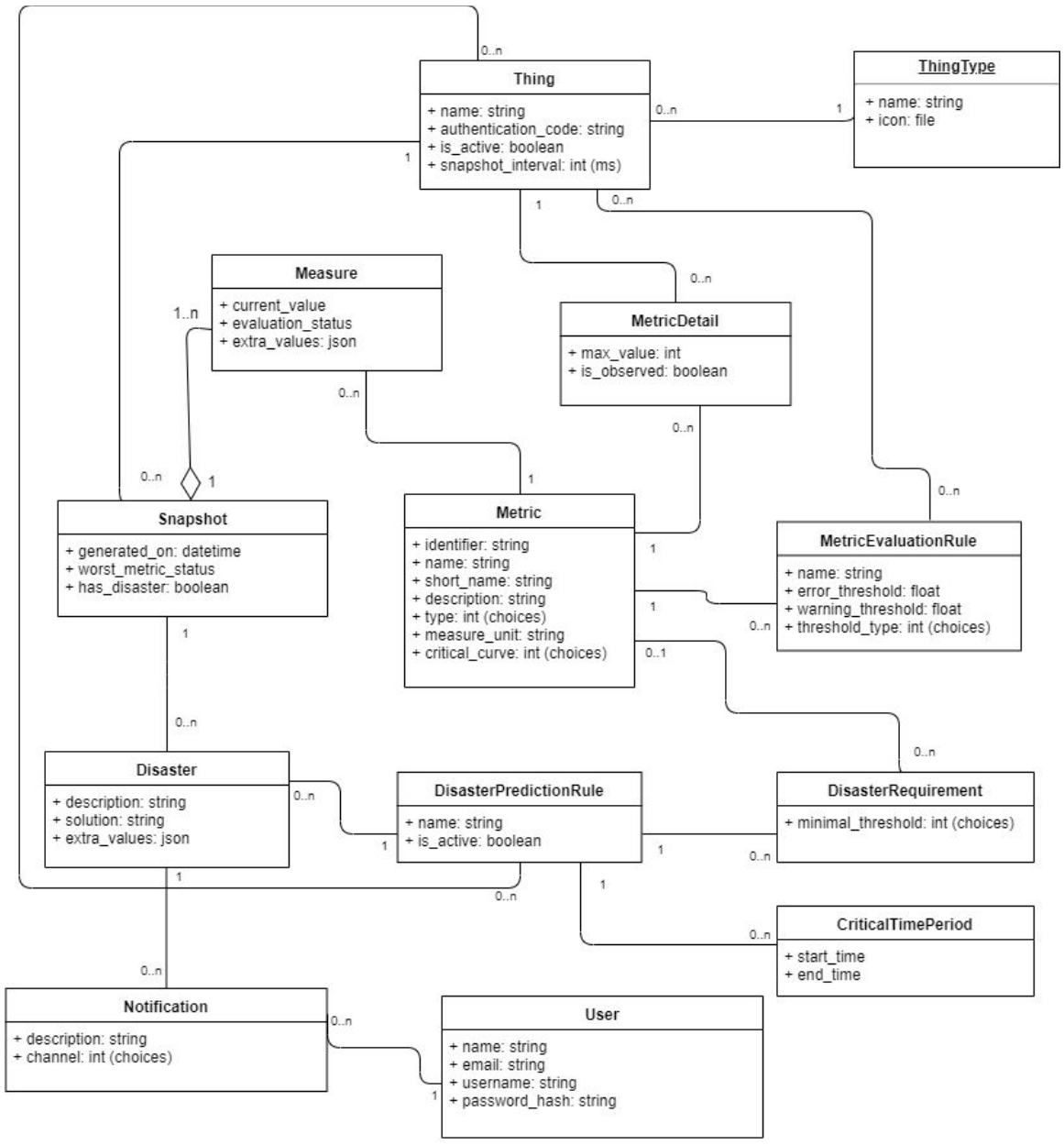


Figura 16. Relacionamento das entidades do sistema

4.3. Arquitetura

Nesta seção veremos como o projeto foi estruturado para entendermos o fluxo dos dados.

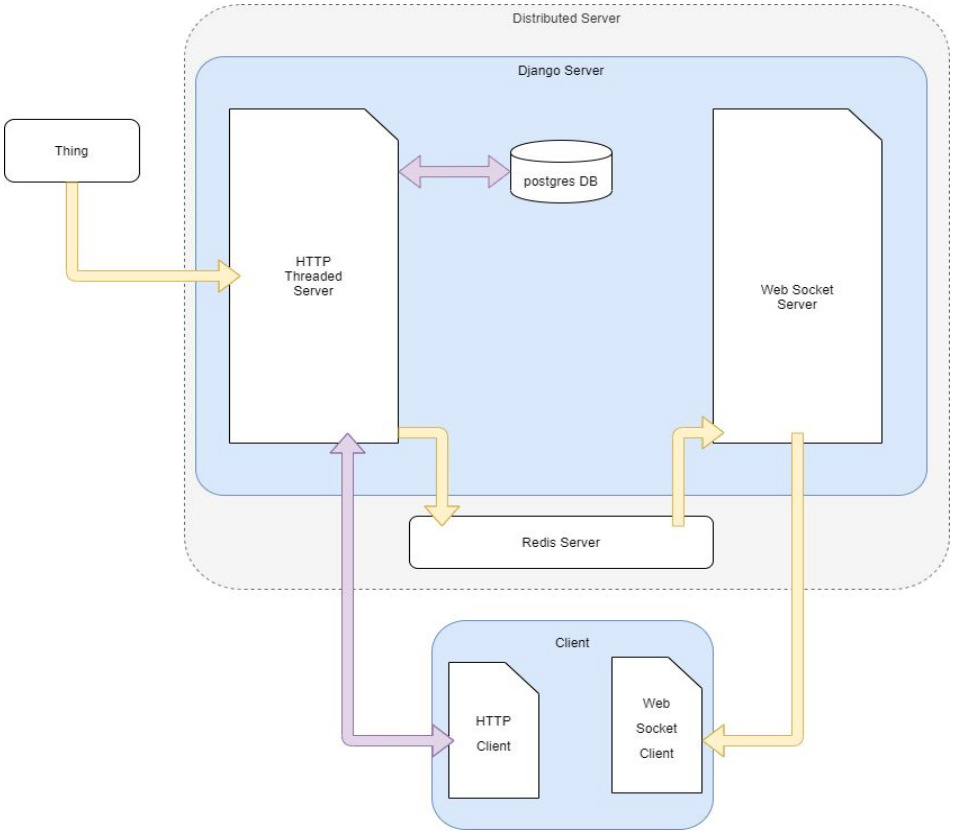


Figura 17. Arquitetura da solução

Quando a "coisa" envia suas métricas para o servidor via API HTTP [12], o servidor que processa essa informação executa duas ações, uma é persistir um Snapshot dessas métricas no banco de dados, outra é disponibilizá-lo para o Redis Server (*cache manager*) [13] que, por sua vez, repassa os dados para o *websocket* do servidor. Dessa forma, o *websocket* [14] cliente poderá receber tais informações sem a necessidade de atualização da tela ou intervenção do usuário para buscar os dados ativamente. É desta maneira que a tabela principal do *dashboard* é atualizada em tempo real de forma automática. Há, ainda, a comunicação ativa disparada pelas requisições do cliente HTTP [12], nesse caso, os dados recebidos por ele são os que já foram persistidos.

4.4. Comunicação para fornecimento de medidas

Nesta seção veremos como se dá a comunicação entre a "coisa" e o servidor que alimenta o *dashboard*. Foram construídas APIs para os dois momentos de comunicação. Todas as comunicações são feitas através do protocolo HTTP [12], utilizando a arquitetura REST [15] de forma a contemplar as melhores práticas no desenvolvimento de aplicações *web*.

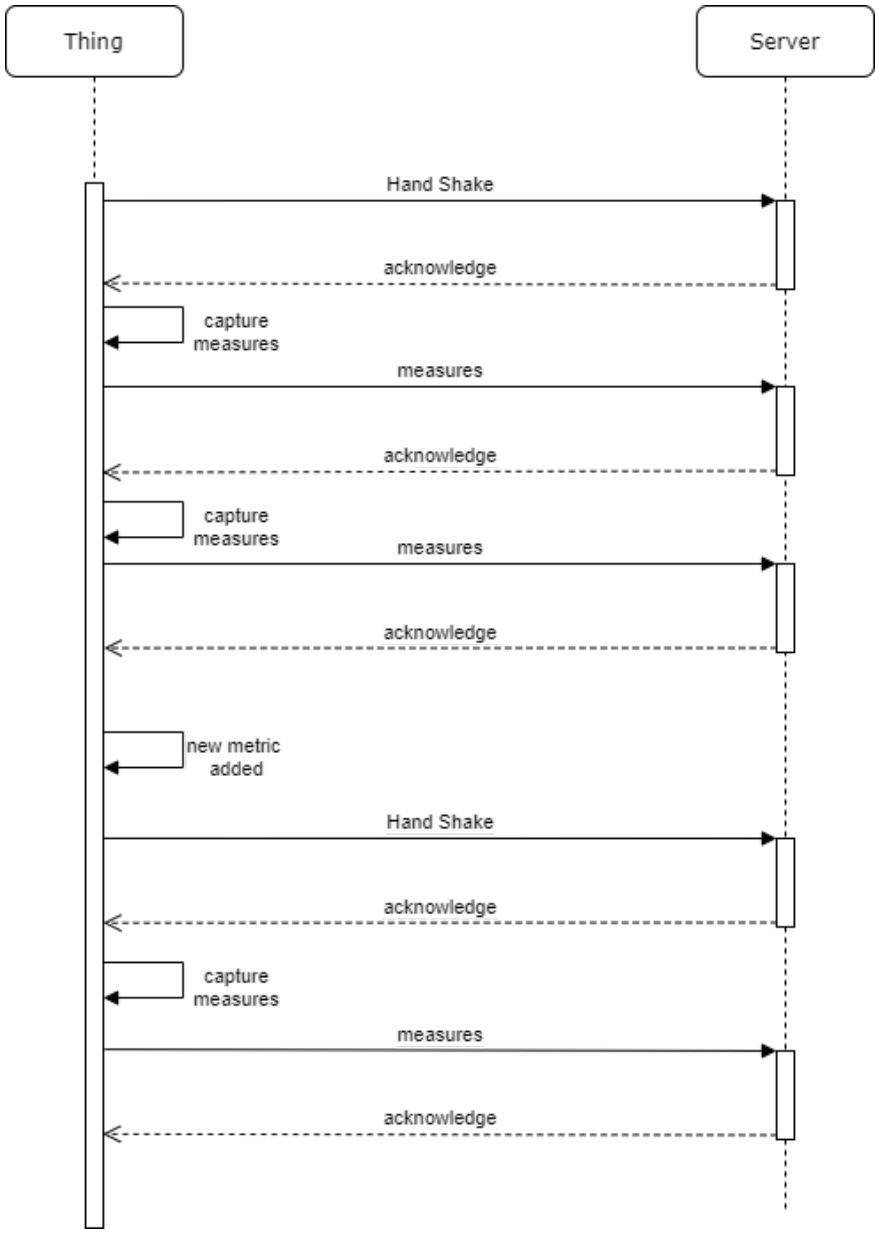


Figura 18. Sequência de comunicação

4.4.1. Handshake

O primeiro passo é o processo no qual a coisa manda, via API HTTP [12], as informações básicas para autenticação, antes de enviar qualquer métrica (já coletada ou não, depende da implementação do fabricante da coisa). Nesta chamada, são enviados as métricas com seus respectivos metadados, a informação da periodicidade com que serão enviadas manda as medidas e o id único da“coisa”. Cabe lembrar que o cadastro prévio já deve ter sido feito no *dashboard*, o que gerou o UUID (*Universally Unique IDentifier*) [16], que é usado pela coisa. A necessidade de *handshake* se dá, portanto, na primeira comunicação ou sempre que uma nova métrica for adicionada, uma vez que o servidor precisa cadastrá-la e que isso acontece em um *handshake*.

```
{
  "authentication_code": "1132b2e8-f979-4194-a621-1c51b534054e",
  "snapshot_interval": 5,
  "metrics": [
    {
      "name": "Memory usage",
      "short_name": "memory usage",
      "description": "Metric gives the usage of the server\'s memory",
      "metric_type": "numeric",
      "mesure_unit": "MBs",
      "critical_curve": "linear_increase",
      "identifier": "memory_metric",
      "max_value": 4000
    },
    {
      "name": "Storage Usage",
      "short_name": "storage usage",
      "description": "Metric gives the usage of the server\'s memory",
      "metric_type": "numeric",
      "mesure_unit": "GBs",
      "critical_curve": "linear_increase",
      "identifier": "storage_metric",
      "max_value": 1000
    },
    {
      "name": "Database Monitor",
      "short_name": "DB monitor",
      "description": "Metric return when the database connection isn\'t working",
      "metric_type": "boolean",
      "mesure_unit": null,
      "critical_curve": "linear_increase",
      "identifier": "database_monitor"
    },
    {
      "name": "Network Usage",
      "short_name": "Net Usage",
      "description": "Metric gives the usage of the server\'s network",
      "metric_type": "numeric",
      "mesure_unit": "Mb/s",
      "critical_curve": "linear_increase",
      "identifier": "network_metric",
      "max_value": 1000
    }
  ]
}
```


Figura 19. Exemplo da estrutura JSON [14] da comunicação no "*handshake*"

Após o envio das informações do *handshake*, o sistema irá verificar as métricas que já foram cadastradas. Caso o sistema não tenha o registro de alguma métrica, esta será adicionada e terá um número de identificação registrado (uma vez que quem envia as métricas já foi validado). Com todas as informações agora cadastradas, o sistema retorna uma lista das métricas habilitadas a receberem medidas e seus respectivos números de

identificação. Esses números serão usados posteriormente para o envio das medidas com o objetivo de identificar a que métricas a medida está relacionada.


```
{
  "success": true,
  "metrics": [
    {
      "identifier": "memory_metric",
      "metric_id": 5
    },
    {
      "identifier": "storage_metric",
      "metric_id": 6
    },
    {
      "identifier": "database_monitor",
      "metric_id": 7
    },
    {
      "identifier": "network_metric",
      "metric_id": 8
    }
  ],
  "error_msg": null
}
```


Figura 20. Exemplo de JSON [14] retorno do "*handshake*"

4.4.2. Envio de medidas

Após o sucesso do *handshake*, o servidor agora sabe quais medidas processar, medidas essas que serão agrupadas em *Snapshots*. A mensagem que é enviada para o sistema deve conter o id único (que identifica quem está mandando informações para *dashboard*), a data em que as medidas foram capturadas e uma lista das medidas. Cada medida deve conter o identificador da métrica e o valor que foi gerado pela coisa.

```
{
  "authentication_code": "1132b2e8-f979-4194-a621-1c51b534054e",
  "generated_on": "2020-12-18T21:17:49.982998-03:00",
  "metrics": [
    {
      "metric_id": 5,
      "identifier": "memory_metric",
      "value": 911
    },
    {
      "metric_id": 6,
      "identifier": "storage_metric",
      "value": 868
    },
    {
      "metric_id": 7,
      "identifier": "database_monitor",
      "value": true
    },
    {
      "metric_id": 8,
      "identifier": "network_metric",
      "value": 307
    }
  ]
}
```


Figura 21. Exemplo da estrutura JSON [14] da comunicação no envio de medidas

4.5. Desafios e recompensas

Sabíamos que a definição dos requisitos era muito importante para reduzirmos os custos do projeto e ficamos satisfeitos com o resultado da etapa anterior. Porém, o processo apresentado neste capítulo era também muito importante e apresentou maiores dificuldades para a tomada de decisões. Foram necessárias algumas versões da modelagem das entidades, uma vez que fomos percebendo, de maneira iterativa, como cada relacionamento ou modelo tornava possível, ou não, uma funcionalidade futura. Isso significa que fizemos um grande esforço para que o modelo atendesse não apenas as funcionalidades cuja construção fosse executada no escopo desse trabalho, mas que também fosse capaz de comportar evoluções futuras.

Outro grande desafio foi imaginar como poderíamos organizar a arquitetura do *software* de forma a possibilitar a exibição do painel de controle com atualizações em tempo real. Apesar de ser um passo bastante delicado para o funcionamento do sistema, foi

menos custoso que o citado acima, mas cujo resultado foi igualmente importante. O planejamento feito nesta etapa e o diagrama produzido facilitaram bastante a implementação posterior.

Capítulo 5 – Organização do código e exemplos

Neste capítulo vamos ilustrar alguns trechos do código fonte, comentando alguns pontos interessantes em relação à implementação.

5.1. Árvore do projeto

A imagem abaixo representa a organização do código do projeto. No nível mais externo, fica o divido entre o sistema do *dashboard*, suas dependências (bibliotecas externas) e o projeto do simulador (responsável por prover dados para o *dashboard*). Além disso, existem outros arquivos de configuração do repositório e variáveis de ambiente.

No próximo nível, dentro do projeto do *dashboard*, temos a aplicação em Django [8] e a configuração das dependências da tabela da tela principal do sistema, feita usando o framework React [10].

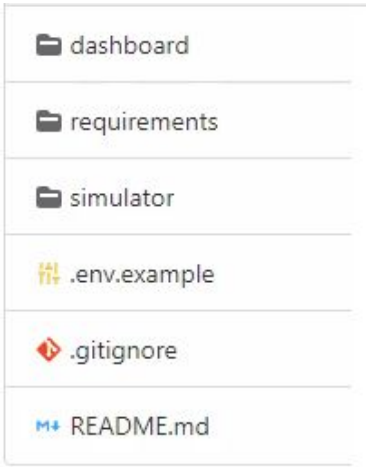


Figura 22: Nível mais externo

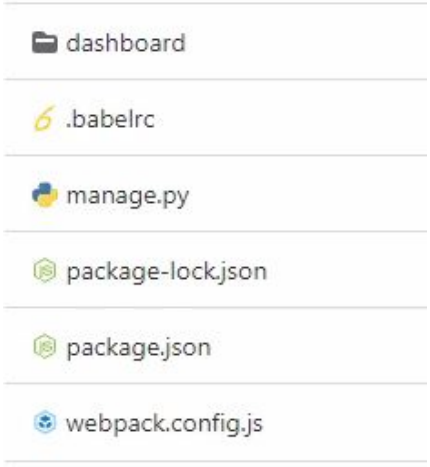


Figura 23: principal app django ("*dashboard*")

Dentro da aplicação do *dashboard*, podemos ver três principais grupos: *backend*, responsável pela lógica de negócio do sistema e por prover APIs para conexão das "coisas"; *frontend*, responsável por prover as telas, visualização dos dados e interação com o usuário; e common, responsável pela infra-estrutura do sistema, como a comunicação com banco de dados e com outros sistemas.

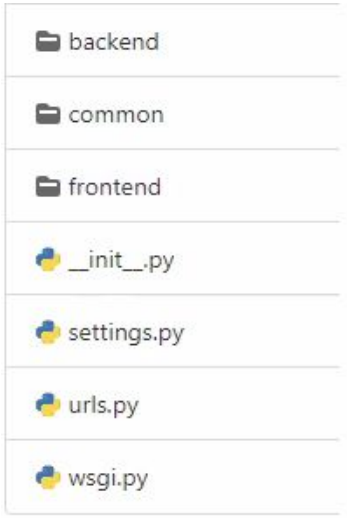


Figura 24: *app* "*dashboard*"

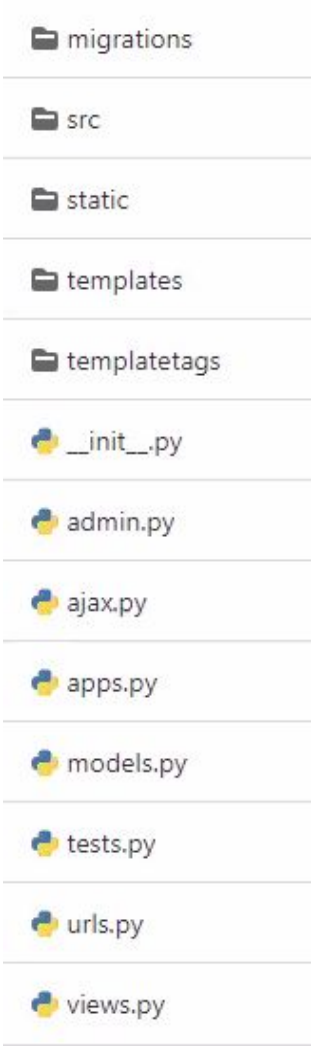


Figura 25: visão do "*frontend*"

O *backend* é dividido em partes, de forma que cada uma é responsável pelos seus próprios módulos e regras de negócio a serem seguidas. Cada parte representa um "*app*" para o *framework* Django [8], dentro da cada "*app*" do *backend*, temos os seguintes itens:

- models.py: Contém a representação da tabela de banco de dados em uma classe
- migrations: Pasta que contém todas as migrações referentes a alterações nos modelos
- admin.py: Contém a especificação dos cadastro de cada modelo
- choices.py: arquivo que contém constantes da aplicação

- api.py: Contém as APIs e regras de negócio específicas do *app*

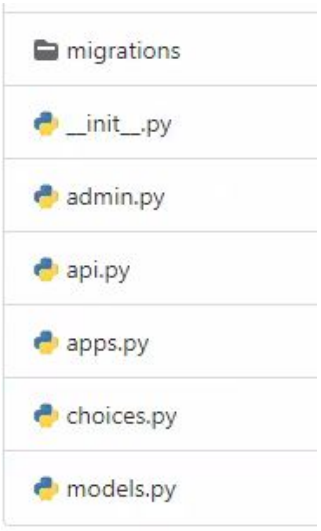


Figura 26: *app* "metric"

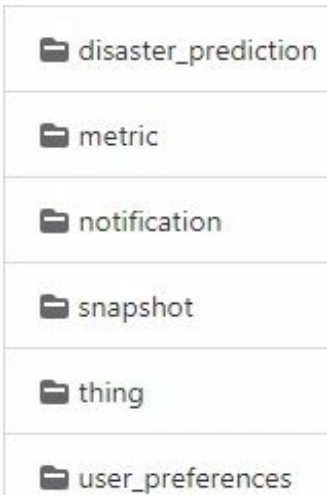


Figura 27: principais *sub-apps* do *backend*

5.2. Exemplos do código

O código abaixo exemplifica a declaração de um modelo que dará origem a uma tabela de banco de dados. Nos modelos, utilizamos um framework de cache que permite que o retorno de chamadas ao banco de dados sejam também mantidas em memória, o que proporciona mais agilidade nas consultas que já foram feitas previamente.

```
class Metric(CachingMixin, models.Model):
    identifier = models.CharField(max_length=200, db_index=True, verbose_name=u"identifier")
    name = models.CharField(max_length=200, verbose_name=u"name")
    short_name = models.CharField(max_length=200, blank=True, null=True, verbose_name=u"short name")
    description = models.TextField(max_length=500, blank=True, null=True, verbose_name=u"description")
    measure_unit = models.CharField(max_length=50, blank=True, null=True, verbose_name=u"measure unit")
    critical_curve = models.IntegerField(choices=CRITICAL_CURVES, verbose_name=u"critical curve")
    metric_type = models.IntegerField(choices=METRIC_TYPES, verbose_name=u"metric type")
    things = models.ManyToManyField(Thing, through='MetricDetail', blank=True, verbose_name=u"things")

    objects = CachingManager()

    class Meta:
        base_manager_name = 'objects'

    def __str__(self):
        return u"%s" %  self.name


class MetricEvaluationRule(CachingMixin, models.Model):

    metric = models.ForeignKey(Metric, on_delete=models.PROTECT, verbose_name="metric")
    name = models.CharField(max_length=200, verbose_name=u"name")
    threshold_type = models.IntegerField(choices=THRESHOLD_TYPES, verbose_name="threshold type")
    error_threshold = models.DecimalField(max_digits=DB_MAX_DIGITS, decimal_places=DB_DECIMAL_PLACES,
                                          verbose_name=u"error threshold")
    warning_threshold = models.DecimalField(max_digits=DB_DECIMAL_PLACES, decimal_places=DB_DECIMAL_PLACES,
                                            verbose_name=u"warning threshold", null=True, blank=True)
    things = models.ManyToManyField(Thing, blank=True, verbose_name=u"things")

    objects = CachingManager()

    class Meta:
         base_manager_name = 'objects'

class MetricDetail(CachingMixin, models.Model):
    metric = models.ForeignKey(Metric, on_delete=models.CASCADE, verbose_name=u"metric")
    thing = models.ForeignKey(Thing, on_delete=models.CASCADE, verbose_name=u"thing")
    max_value = models.DecimalField(max_digits=DB_MAX_DIGITS, decimal_places=DB_DECIMAL_PLACES,
                                    verbose_name=u"max matric value", blank=True, null=True)
    is_observed = models.BooleanField(default=False, verbose_name=u"metric is observed")

    objects = CachingManager()

    class Meta:
        base_manager_name = 'objects'
```

Figura 28: principais classes do *app* "*metric*"

As duas figuras abaixo mostram como o sistema executa a avaliação de medidas recebidas.

Para cada medida recebida, é identificada a regra de avaliação que deve ser usada e, a partir dela, que parâmetros servirão de base. Caso o sistema não encontre uma regra para avaliar a medida, é assumido que ela está dentro dos limites.

O processo de avaliação verifica, inicialmente, qual o tipo da curva de crescimento da medida, se ela é linearmente crescente (o nível de criticidade aumenta de acordo com o crescimento do valor medido de forma linear) ou se é linearmente decrescente. Após verificar essa informação, o sistema usa os limites de “erro” e “aviso” para atribuir um resultado de avaliação à medida.

```
def evaluate_measures(thing_id, measures, metric_ids=None, metrics_info=None, metric_details=None):
    """
    evaluate the measures based on the metric evaluation rule that matches with the thing_id and the metric_id
    """

    data = {}
    if not measures or not thing_id:
        return data

    if not metric_ids:
        metric_ids = [m['metric_ids'] for m in measures]

    filters = {
        'metric_id__in': metric_ids,
    }
    complex_filter = Q(things__in=[thing_id]) | Q(things=None)

    evaluation_rules_qs = MetricEvaluationRule.objects.filter(complex_filter, **filters)

    evaluation_rule_dict = {}

    for ev in evaluation_rules_qs:
        metric_id = ev.metric_id
        if not len(ev.things.all()):
            key = (None, metric_id)
            evaluation_rule_dict.setdefault(key, []).append({'threshold_type': ev.threshold_type,
                                                             'error_threshold': ev.error_threshold,
                                                             'warning_threshold': ev.warning_threshold})
            continue
        for ev_thing in ev.things.all():
            key = (ev_thing.thing_id, metric_id)
            evaluation_rule_dict.setdefault(key, []).append({'threshold_type': ev.threshold_type,
                                                             'error_threshold': ev.error_threshold,
                                                             'warning_threshold': ev.warning_threshold})
    if not metrics_info:
        metrics_info = get_metric_info(ids=metric_ids)

    if not metric_details:
        metric_details = get_metric_detail_info(thing_ids=[thing_id], metric_ids=metric_ids)
    md_by_thing_metric_id = {(md.get('thing_id'), md.get('metric_id')): md for md in metric_details.values()}

    def make_rule_key(combiation, rule_params):
        rule_key = []
        for i, value in enumerate(combiation):
            rule_key.append(rule_params[i] if value else None)
        return tuple(rule key)
```

Figura 29: exemplo de função do app “metric” (parte 1)

```
            rule_key.append(rule_params[i] if value else None)
        return tuple(rule_key)

    for measure in measures.values():
        metric_id = measure['metric_id']
        current_value = measure['current_value']
        evaluation_rule_parameters = [thing_id, metric_id]
        possible_combinations = sorted(list(product([0,1], repeat=len(evaluation_rule_parameters))))
        evaluation_rules = []
        for c in possible_combinations:
            evaluation_rule_key = make_rule_key(c, evaluation_rule_parameters)
            evaluation_rules = evaluation_rule_dict.get(evaluation_rule_key, [])
            if evaluation_rules:
                break
        data[metric_id] = measure.copy()
        if not evaluation_rules:
            data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_all_good
            continue
        evaluation_rule = evaluation_rules[0] # getting the first
        metric_info = metrics_info.get(metric_id, {})
        # by default critical curve will be increase
        critical_curve = metric_info.get('critical_curve', CRITICAL_CURVES.linear_increase)
        warning_threshold = evaluation_rule.get('warning_threshold')
        error_threshold = evaluation_rule.get('error_threshold')
        metric_detail = md_by_thing_metric_id.get((thing_id, metric_id), {})
        if evaluation_rule.get('threshold_type') == THRESHOLD_TYPES.percentual:
            if not metric_detail.get('max_value'):
                raise Exception("System couldn't evaluate the measure cause there is no detailed information.")
            current_value = Decimal(current_value) / Decimal(metric_detail['max_value'])
        if critical_curve == CRITICAL_CURVES.linear_increase:
            if current_value >= error_threshold:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_error
            elif current_value >= warning_threshold:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_warning
            else:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_all_good
        elif critical_curve == CRITICAL_CURVES.linear_decrease:
            if current_value <= error_threshold:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_error
            elif current_value <= warning_threshold:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_warning
            else:
                data[metric_id]['evaluation_status'] = MEASURE_EVALUATION_STATUS.metric_all_good
        else:
            data[metric_id]['evaluation_status'] = CRITICAL_CURVES.linear_increase
    return data
```

Figura 30: exemplo de função do app “metric” (parte 2)

Para fazer a comunicação entre o *frontend* e o *backend*, criamos funções no arquivo “ajax.py” do *frontend*. Eles são responsáveis pela exposição de algumas APIs do *backend* e tornam possível a consulta de informações feitas pela tabela da tela principal do *dashboard*.

O exemplo abaixo, é usado para consultar os detalhes de um desastre.

```python
@require_http_methods(["POST"])
@csrf_exempt
@login_required
def get_disaster_details(request):
    try:
        request_data = json.loads(request.body)
        disaster_id = request_data.get('disaster_id')
        if not disaster_id:
            raise Exception("disaster_id required")
        disaster_info = disaster_api.get_disaster_info(disaster_ids=[disaster_id]).get('disasters', {})\
                        .get(disaster_id, {})
        snapshot_id = disaster_info.get('snapshot_id')
        snapshot_detail = snapshot_api.get_last_snapshots(snapshot_ids=[snapshot_id], by_snapshot_id=True).get(snapshot_id)
        data = {
            'success': True,
            'snapshot_detail': snapshot_detail,
            'disaster_detail': disaster_info,
        }
        return HttpResponse(json.dumps(data, cls=ExtendedJSONEncoder), content_type="application/json")
    except Exception as e:
        response_data = {'success': False, 'error_msg': str(e)}
        return HttpResponse(json.dumps(response_data), content_type="application/json")
```

Figura 31: exemplo de função ajax

No *frontend*, utilizamos o *framework* de tabela ag-Grid [11], desenvolvido em React [10], para construir uma tabela responsiva e ágil, com capacidade de exibir dados em tempo real. O código abaixo demonstra a inicialização da tabela e como o processo de alimentação de dados é feito.

Ao carregar a página, o sistema carrega os dados iniciais da última fotografia das medidas das coisas. Isso significa que os dados apresentados ao abrir a página são provenientes do banco de dados. Após a inicialização, a comunicação via *Socket* [14] começa a valer, por onde serão enviadas as medidas e suas avaliações em tempo real. A cada nova medida recebida, o sistema realiza a contagem das medidas que estão com “erro”, “alerta” e “normal”, avaliações que serão usadas no resumo lateral do *dashboard*.

```
onGridReady(params) {
  this.gridApi = params.api;
  this.gridColumnApi = params.columnApi;
  if (window.USER_PREFERENCES.dashboard_preferences) {
      this.gridColumnApi.setColumnState(window.USER_PREFERENCES.dashboard_preferences);
  }
  const { thing_ids, metric_ids, get_snapshot_url, reloadOnlineColumnTime } = this.state;
  const body = {thing_ids: thing_ids, metric_ids: metric_ids}
  fetch(get_snapshot_url, {
      'method': 'post',
      'body': JSON.stringify(body)
  })
  .then((res) => res.json())
  .then(res => {
      let counters = {error: 0, warning: 0, good: 0};
      const firstData = Object.values(res).map((snapshot) => {
          const thingId = snapshot.thing_id;
          const newSnapshot = {
              ...this.evaluateSnapshot(snapshot),
              thing_name: window.ACTIVE_THINGS[thingId].name,
              thing_type_name: window.ACTIVE_THINGS[thingId].thing_type__name,
          };
          counters.error += newSnapshot.counters.error;
          counters.warning += newSnapshot.counters.warning;
          counters.good += newSnapshot.counters.good;
          return newSnapshot;
      });
     return this.setState({rowData: firstData, counters});
  })
  .then(() => setTimeout(setInterval(this.reloadOnlineColumn, reloadOnlineColumnTime), 10000))
  .catch(err => alert(err.toString()));
  window.webSocket.subscribe('snapshot_data', this.processNewSnapshot);
}
```

Figura 32: tabela principal do *dashboard*

Para obtermos dados que a serem avaliados pelo *dashboard*, criamos um simulador, uma aplicação paralela, que realiza o processo de autenticação e de envio das medidas respeitando o protocolo determinado pelo *dashboard*. Cada simulador possui suas próprias configurações e realiza o *handshake* e envio das medidas de maneira independente.

Abaixo, podemos ver como o simulador implementa o *handshake*, enviando também as informações necessárias para informar quais métricas terão medidas enviadas ao longo da comunicação. Além disso, é possível ver como as medidas são geradas, de acordo com suas características, e como elas são enviadas.

```
class Simulator:
    identifier = None
    metrics = {}
    snapshot_interval = None
    scheduler = sched.scheduler(time.time, time.sleep)

    def __init__(self, identifier, metrics, snapshot_interval):···

    def send_handshake(self, data):···

    def handshake(self):
        """
        send all worker identification and metrics that will be sent.
        "handshake": {"metrics": ["metric"], "authentication_code" : string}
        "metric": {"name": string, "short_name": string, "description": string,
                "metric_type": choices("number", "boolean"),
                "metric_unit": string or None, 'identifier': string}
        """
        data = {"authentication_code": self.identifier, "snapshot_interval": self.snapshot_interval,
                "metrics": []}
        for metric_id, metric in self.metrics.items():
            if metric.meta_data:
                send_metric = metric.meta_data.copy()
                send_metric['identifier'] = metric_id
                if metric.max_value:
                    send_metric['max_value'] = metric.max_value
                data['metrics'].append(send_metric)
        if data['metrics']:
            handshake_result = self.send_handshake(data)
            if handshake_result.get('success'):
                for dm in handshake_result['metrics']:
                    if self.metrics.get(dm['identifier']):
                        self.metrics[dm['identifier']].set_metric_id(dm['metric_id'])
            return handshake_result
        else:
            raise Exception("no metrics were found")

    def start_generate_data(self):···

    def generate_metrics(self, scheduler):
        """
        generate metrics
        """
        print("Generate metrics... %s" % self.identifier)
        generated_data = {'generated_on': None, 'metrics': []}
        for metric in self.metrics.values():
            metric_data = metric.generate_metric_data()
            generated_data['metrics'].append(metric_data)
        if generated_data['metrics']:
            generated_data['generated_on'] = datetime.datetime.now()
            self.send_metrics_data(generated_data)
        self.scheduler.enter(self.snapshot_interval, 1, self.generate_metrics, (scheduler,))
```

Figura 33: simulador que gera dados para alimentar a tabela

Capítulo 6 – O sistema e avaliação crítica

Neste capítulo vamos ilustrar os principais casos de uso do sistema e fazer uma análise da solução proposta, destacando os pontos positivos e possíveis melhorias.

6.1. Ilustração

Como pode ser visto na figura a seguir, a página principal do sistema contempla diversas funcionalidades do MVP 1 de forma bem imediata:

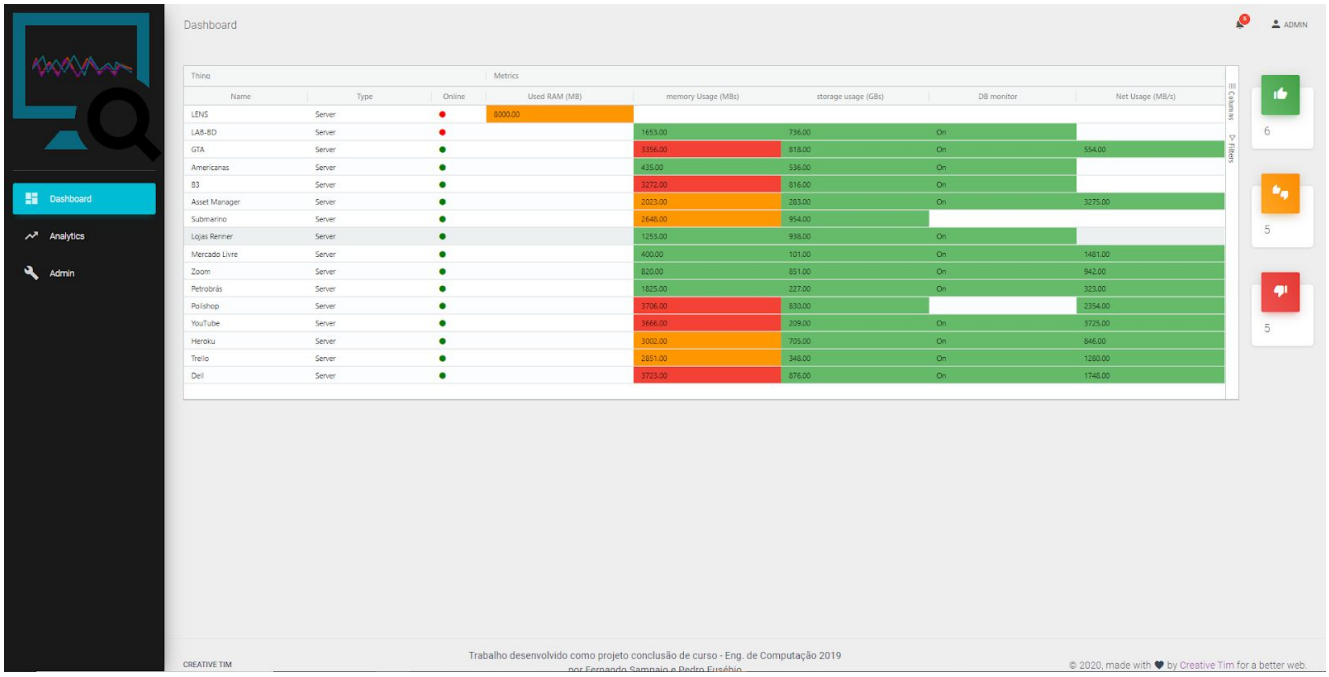


Figura 34. Página principal, *dashboard* de controle (F22: Manter *dashboard*)

À esquerda, menus de acesso à página principal do *dashboard*; ao painel de relatórios (“Analytics”), que seria objeto de construção em MVP 3 futuro; e ao painel de administração do sistema (“Admin”), onde são cadastradas as “coisas” monitoradas e todos os parâmetros desejados (tais quais como “limites”, “usuários do sistema” etc). Agora, vamos focar em cada informação do *dashboard* e como elas refletem a implementação do uma ou mais funcionalidade(s) dos MVPs 1 e 2 construídos:

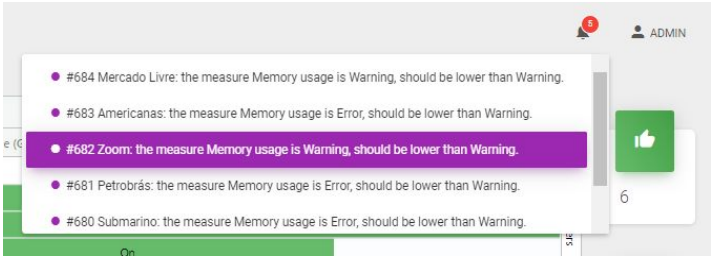


Figura 35. Notificações (Funcionalidade F1 “Sistema de notificação no dashboard”)

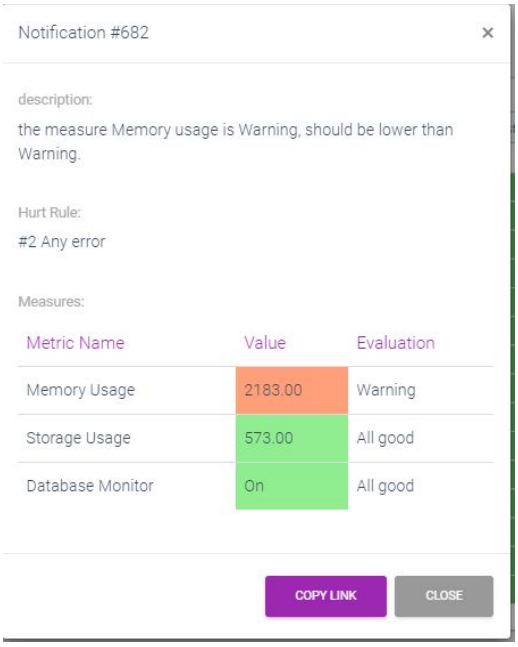


Figura 36:. Detalhe da notificação (Funcionalidades F18: Armazenar snapshots do status das métricas coletada; F6: Detalhamento em relação aos processos/serviços/recursos monitorados; F2: Categorização em graus de criticidade; F23: Compartilhar incidente; F21: Obter dados dos recursos físicos do servidor)

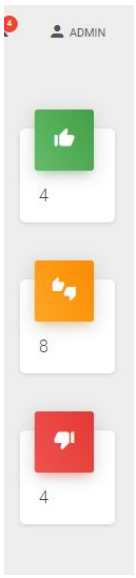


Figura 37. Resumo da situação (F7: Seção “Resumo” no *dashboard* de *health check*)

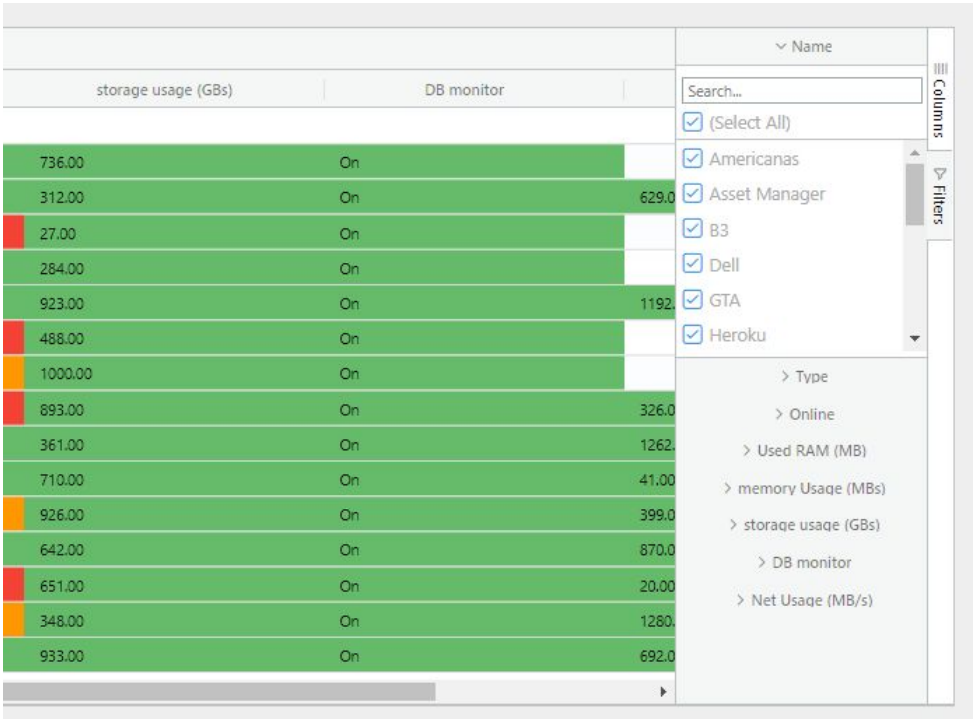


Figura 38. Recursos da tabela do *dashboard* (F13: *Dashboard* ajustável para visualização segmentada por aquilo que interessa; F12: Agrupar incidentes por status/sintomas/grau de criticidade (funcionalidades foram agrupadas em um único item)

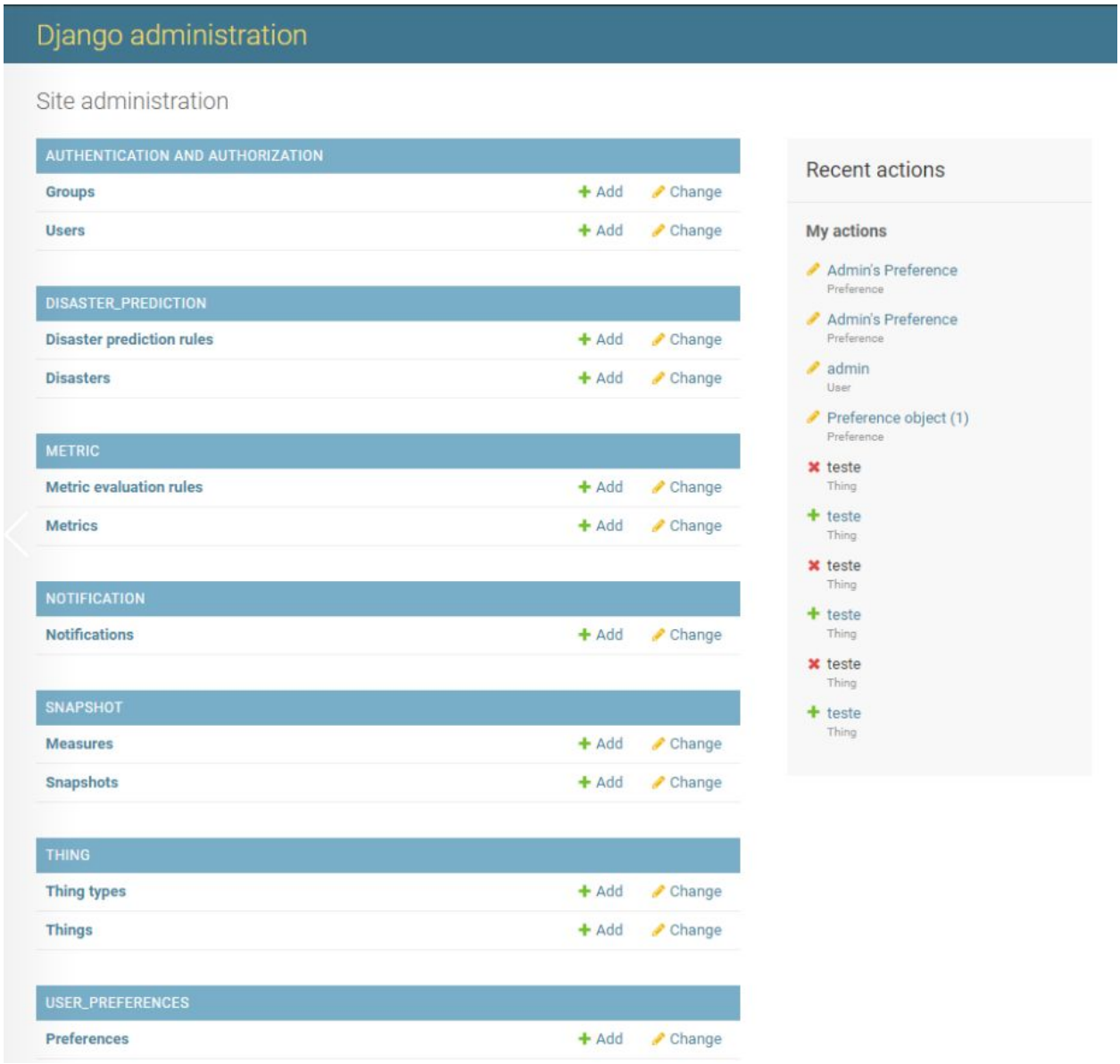


Figura 39. Seção “Admin” (F22: Manter *dashboard*; F4: Predição de ocorrência de falhas; F18: Armazenar snapshots do status das métricas coletadas; F17: Ter segmentação por tipos de problema (“negócio”, “infra”...) no *dashboard*; F6: Detalhamento em relação aos processos/serviços/recursos monitorados)

6.2. Exemplo de jornada implementada

Apesar de sabermos que as jornadas de cada persona ficarão cada vez mais bem atendidas conforme o sistema evoluir, já podemos perceber como, por exemplo, a jornada 2 (do "Ricardinho Plantonista" já se encontra plenamente atendida. O primeiro passo dela (verificação de existência de problema crítico) é atendido pela seção resumo (figura 34) em conjunto com as notificações do sistema (figura 32). O passo seguinte (alertar um amigo da equipe) é atendido pelo compartilhamento de notificações (figura 33). Por fim, o acompanhamento da situação geral é atendido pela tabela principal do *dashboard* (figura 31).

6.3. Pontos positivos

Entendemos que a proposta de um visual "limpo" e objetivo foi atendida. Além disso, como pudemos destacar, é fácil perceber que todas as funcionalidades dos MVPs construídos, 1 e 2, são facilmente identificadas. Isso faz com que o usuário final do *dashboard* tenha uma experiência prazerosa e com boa usabilidade [17].

6.4. Possíveis melhorias

Apesar de termos um sistema bastante intuitivo, a manutenção (seção "Admin") precisaria de um aspecto mais amigável. Atualmente possui a "cara" provida pelo framework de persistência Django [8], em uma etapa mais avançada do projeto, portanto, seria necessária a personalização da área de administração, bem como a inclusão de instruções de uso para a equipe de manutenção do *dashboard*.

Além disso, queremos aumentar a quantidade de personalizações que podem ser feitas no *frontend* do sistema de acordo com o usuário. Esse tipo de poder tornará a experiência mais personalizada, o que aumenta o grau de satisfação em relação ao uso do sistema.

Capítulo 7 – Disposições Finais

Neste capítulo apresentamos nossa conclusão acerca do trabalho realizado e pontuamos os próximos passos para continuarmos evoluindo o sistema construído, além de reforçar a motivação dessa evolução.

7.1. Conclusão

Neste trabalho foi apresentado um painel de controle (*dashboard)* de sistemas autônomos capaz de se comunicar a partir de um protocolo próprio. Foram apresentados o cenário envolvido, a motivação do trabalho, trabalhos correlatos, a tecnologia utilizada, sua arquitetura e funcionamento geral, assim como as vantagens e deficiências do projeto. Algo que vale grande destaque foi a aplicação da metodologia "*Lean Inception*" [3], explicitada no capítulo 3. Através dela, tivemos grande facilidade de identificar o que deveria, de fato, ser construído. Além disso, a tarefa mostrou-se "de baixo custo mental", no sentido de que a simples aplicação da metodologia fez com que chegássemos às conclusões com o esforço reduzido e, ainda, com grande segurança de que estávamos no caminho certo.

O resultado trouxe grande satisfação, principalmente, por se mostrar preparado para evoluir sem que tenhamos necessidade de grandes mudanças na modelagem atual. Isso mostrou como podemos, inclusive, mudar nosso processo profissional para incorporar a metodologia aqui aplicada quando nos depararmos com um grande desafio.

7.2. Trabalhos futuros

Como pontuamos no início, nossa intenção é usar o sistema aqui construído para nos auxiliar em tarefas do nosso cotidiano profissional. Por isso, para que este *dashboard* fique ainda mais completo e traga ainda mais benefícios às nossas rotinas, seria interessante a construção do MVP 3, que contempla as seguintes funcionalidades, já divididas por onda de desenvolvimento:

- Onda de desenvolvimento 5:

    - F3: Parametrização do sistema de notificação de acordo com o grau de criticidade
    - F10: Manter lista de incidentes que estão sendo tratados (com responsável)
    - F20: Sistema de notificações externas ao *dashboard*
- Onda de desenvolvimento 6:
    - F9: Módulos de relatórios de consumo e de incidentes

Aqui nos cabe ressaltar mais um grande ganho que a *Lean Inception* nos trouxe: em um processo menos organizado, poderíamos ter avaliado a funcionalidade F3, acima, como essencial para uma primeira versão do produto, no entanto, não foi o que concluímos. Apenas foi essencial preparar a modelagem de forma a, não apenas ter segregação por grau de criticidade, mas também, poder configurar o sistema de notificação de acordo. Entretanto, para a versão inicial do produto, esse passo consegue ser executado manualmente, uma vez que é possível compartilhar o incidente. Com isso, o “Ricardinho Plantonista” é capaz de alertar o "Joãozinho Sherlock” sobre os problemas críticos. Esse tipo de clareza foi fundamental para a construção de uma solução coerente com o prazo pretendido e preparada para as evoluções desejadas.

Referências

[1] SILVA, Valéria Martins da. *Scenariot: Support For Scenario Specification Of Internet Of Things-based Software Systems*. M. Sc. Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 2019.

[2] EBERT, C., GALLARDO, G., HERNANTES, J., SERRANO, N.: Devops. IEEE Softw. 33(3), 94–100 (2016)

[3] CAROLI, Paulo. ***Lean Inception: Como Alinhar Pessoas E Construir O Produto Certo***. 1 ed. Porto Alegre: Loope Editora, 2018.

[4] BRAGA, Igor & NOGUEIRA, Marcelo & SANTOS, Nuno, et al., 2020. "Does the Lean Inception Methodology Contribute to the Software Project Initiation Phase?". *Lecture Notes in Computer Science*. Cagliari, Itália, 1–4 Julho 2020.

[5] LENARDUZZI, Valentina & TAIBI, Davide, 2016, "MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product". 10.1109/SEAA.2016.56.

[6] ANWAR, A.: A review of RUP (Rational Unified Process). Int. J. Softw. Eng. (IJSE) 5(2), 12–19 (2014)

[7] PYTHON 3.7.4 2019. Disponível em: https://www.python.org/downloads/release/python-374/; Acessado em: 4 de Out. 2020 19:43:39.

[8] DJANGO 2.2 2019. Disponível em: https://docs.djangoproject.com/en/3.1/releases/2.2/; Acessado em: 4 de Out. 2020 16:20:61.

[9] POSTGRESQL 11 2018. Disponível em: https://www.postgresql.org/; Acessado em 4 de Out. 2020 18:46:58.

[10] REACT 16.12.0 2019. Disponível em: https://reactjs.org/; Acessado em 4 de Out. 2020 17:33:54.

[11] AG GRID 2019. Disponível em: https://www.ag-grid.com/; Acessado em: 4 de Out. 2020 00:01:10.

[12] HTTP; Disponível em: http://www.ietf.org/rfc/rfc2616.pdf, 1.1, p.7; Acessado em 4 de Out. 2020 12:21:12.

[13] REDIS SERVER; Disponível em: https://redis.io/; Acessado em 4 de Out. 2020 13:33:45.

[14] JUNEAU, Josh. (2020). WebSockets and JSON. 10.1007/978-1-4842-5587-2_14.

[15] Miller, Mark A et al. “A RESTful API for Access to Phylogenetic Tools via the CIPRES Science Gateway.” *Evolutionary bioinformatics online* vol. 11 43-8. 16 Mar. 2015, doi:10.4137/EBO.S21501

[16] LEACH, Paul & MEALLING, Michael & SALZ, R.. (2005). A Universally Unique IDentifier (UUID) URN Namespace.

[17] DHILLON, B., 2019, “Software and web usability”. In: CRC Press, ***Systems Reliability and Usability for Engineers***, 1 ed, chapter 7, 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL, 2019.