UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VINÍCIUS SOUZA SEIXAS DE OLIVEIRA

AUTOFORMVALIDATION: VALIDAÇÃO DINÂMICA BASEADA NOS METADADOS DO BANCO DE DADOS DA APLICAÇÃO

CAMPINA GRANDE - PB

2021

VINÍCIUS SOUZA SEIXAS DE OLIVEIRA

AUTOFORMVALIDATION: VALIDAÇÃO DINÂMICA BASEADA NOS METADADOS DO BANCO DE DADOS DA APLICAÇÃO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Maxwell Guimarães de Oliveira.

CAMPINA GRANDE - PB

2021

O482a Oliveira, Vinícius Souza Seixas de.
AutoFormValidation: validação dinâmica baseada nos metadados do banco de dados da aplicação / Vinícius Souza Seixas de Oliveira. – 2021.

11 f.

Orientador: Professor Dr. Maxwell Guimarães de Oliveira.

Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Sistemas de informação. 2. Metadados. 3. API metadata. 4. AutoFormValidation. 5. Data input. 6. Data validation. 7. Sistema de gerenciamento de banco de dados I. Oliveira, Maxwell Guimarães de. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

VINÍCIUS SOUZA SEIXAS DE OLIVEIRA

AUTOFORMVALIDATION: VALIDAÇÃO DINÂMICA BASEADA NOS METADADOS DO BANCO DE DADOS DA APLICAÇÃO

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Maxwell Guimarães de Oliveira

Orientador – UASC/CEEI/UFCG

Professor Dr. Marcus Salerno de Aquino

Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de maio de 2021.

CAMPINA GRANDE - PB

ABSTRACT

In Information Systems, it is commonly necessary to validate input data according to the rules modeled and implemented into the database in order to keep data integrity and consistency. Developers usually implement input validations in a static way, considering the current model/state of the database. This may imply reimplementation of the validation whenever there are changes in the database. In order to smooth this problem, we developed the AutoFormValidation, which provides dynamic form validation as it relies on the live database metadata. An use case is presented aiming at exemplifying and discussing the usage and utility of the proposed solution. We also highlight dynamic validation prevents the insertion of source code with inconsistencies within the database.

AutoFormValidation: validação dinâmica baseada nos metadados do banco de dados da aplicação

Vinícius Souza Seixas de Oliveira
vinicius.oliveira@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Maxwell Guimarães de Oliveira
maxwell@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

ABSTRACT

In Information Systems, it is commonly necessary to validate input data according to the rules modeled and implemented into the database in order to keep data integrity and consistency. Developers usually implement input validations in a static way, considering the current model/state of the database. This may imply reimplementation of the validation whenever there are changes in the database. In order to smooth this problem, we developed the AutoFormValidation[1], which provides dynamic form validation as it relies on the live database metadata. An use case is presented aiming at exemplifying and discussing the usage and utility of the proposed solution. We also highlight dynamic validation prevents the insertion of source code with inconsistencies within the database.

Keywords

Information Systems, metadata, data input, data validation.

1. INTRODUÇÃO

Sistemas de Informação (SI) [1] costumam ser desenvolvidos com o propósito de gerenciar dados, permitindo a interação do usuário para criar, listar, editar e deletar diversos tipos de registros. Essas operações básicas recebem a denominação CRUD (*Create, Read, Update and Delete*) [2] em um SI. Neste contexto, a inserção e edição de dados ocorre através de formulários, responsáveis por receber os valores informados pelos usuários.

Os formulários de uma aplicação (termo sinônimo a um SI) costumam fazer referência às entidades (ou tabelas) do Sistema de Gerenciamento de Banco de Dados (SGBD) utilizado por esta. Desse modo, é muito comum que os formulários tenham a responsabilidade de validar se os valores informados pelos usuários em seus campos estão de acordo com as regras, conhecidas como *constraints*, definidas e implementadas no SGBD, antes de proceder com o envio dos dados para armazenamento.

As *constraints* [3] de uma entidade do SGBD definem as regras para inserção e edição de cada campo de um formulário. *Constraints* podem, por exemplo, especificar se um campo é de preenchimento obrigatório, o limite máximo de caracteres armazenados, etc. Logo, essas regras pré-estabelecidas demandam que a aplicação valide os valores inseridos pelos usuários, de modo que elas não sejam violadas. A validação de dados é um dos requisitos necessários para manter a integridade dos dados trabalhados [4].

No contexto de uma aplicação, as validações são comumente implementadas estaticamente no código-fonte pelos desenvolvedores, checando cada regra que deve ser validada, o que causa alguns problemas. Além do tempo dedicado ao desenvolvimento desta funcionalidade, torna-se necessário garantir a consistência dos dados em relação às validações. De modo que, qualquer regra que no formulário for implementada diferente do definido nas *constraints* irá gerar erros no instante de armazenar dados da entidade. Como, por exemplo, o usuário não ser alertado de que o campo excedeu o limite de caracteres, recebendo um erro no instante de efetivar o registro do dado.

Mesmo que, durante o desenvolvimento, as validações tenham sido de acordo com as *constraints*, alterações futuras no SGBD podem alterar as regras. Isto gera impacto direto na funcionalidade de validação, sendo necessário uma reavaliação das regras para reimplementá-las, pois o contrário as deixariam inconsistentes. Situações como esta interferem na confiança do usuário em relação a aplicação. A credibilidade é comprometida.

No presente documento, o AutoFormValidation é apresentado como solução para o problema da validação de campos de formulários. O mesmo trabalha dinamizando a validação com o uso de metadados da entidade equivalente no SGBD da aplicação. A abordagem garante a consistência dos dados inseridos, mesmo quando houver alteração das *constraints* da entidade, e sem a necessidade de alterações no código-fonte. A solução foi desenvolvida para fazer a validação no lado do cliente, sendo ativada para cada inserção de caractere em um campo de um formulário.

O AutoFormValidation oferece a garantia de consistência da inserção e edição de dados na aplicação, mesmo quando houver alterações nas *constraints* das entidades no SGBD. Além disso, a solução ajuda a tornar mais eficiente o processo de desenvolvimento de software.

A arquitetura e os artefatos da solução proposta são apresentados na seção 2, bem como detalhes das decisões tomadas e do código-fonte implementado. Na seção 3, apresentamos um caso de uso baseado em um formulário de uma aplicação *web* para demonstrar a validação automática dos campos com base nos metadados da entidade correspondente em seu SGBD. Por fim, na

seção 4, concluímos o trabalho apontando direcionamentos para trabalhos futuros.

O código-fonte, tanto da solução proposta quanto do caso de uso apresentado neste documento, está disponível no seguinte repositório: https://github.com/vsseixaso/AutoFormValidation, juntamente com as instruções para utilização. No repositório, é possível clonar todo o projeto para diferentes fins ou propor a implementação de melhorias e incrementos.

2. SOLUÇÃO DESENVOLVIDA

Para reduzir os problemas associados à validação de campos de formulários em aplicações *web*, ou aplicações que utilizam de linguagens da *web* em seus formulários, foi desenvolvido o AutoFormValidation, um validador automatizado que se baseia nos metadados do SGBD.

2.1 Método

Foi desenvolvida uma API (*Application Programming Interface*), em Node.js [5], responsável por servir os metadados tratados e obtidos através da entidade equivalente no SGBD MySQL [6]. Nomeada ***metadata***, a API recebe no *endpoint* o parâmetro ***tableName***, que indica a entidade do SGBD da qual deve-se obter os metadados dos campos a serem verificados no formulário do cliente. Antes de retornar as regras de validação no formato JSON, a API faz um tratamento eliminando do resultado as propriedades do SGBD que não serão úteis para o processo (ex: TABLE_SCHEMA, TABLE_NAME, ORDINAL_POSITION) [7] e formatando a estrutura dos dados para que fique legível e de fácil utilização.

Para utilizar a *metadata API*, a aplicação *web* precisa fazer uma chamada ao serviço *metadataService* encarregado de obter os metadados com uma requisição HTTP, introduzir um código-fonte JavaScript (JS) que contém as funções de validação baseadas nestes metadados, e fazer a chamada destas funções em seus formulários. Como parâmetro, é passado o valor inserido em um campo e as regras equivalentes ao mesmo. A aplicação *web* que estiver fazendo uso deste serviço pode então, por exemplo, exibir as respostas destas funções ao usuário, como mensagens de erro de validação.

O processo implementado irá, de forma dinâmica e automática, validar os campos de formulários de uma aplicação *web*, tendo como base os metadados da entidade equivalente no SGBD, o que garante de ponta a ponta a consistência dos dados. Dessa forma, com o processo atuando em todas as frentes (formulários) do sistema, um componente de *input* no *front-end* será validado diretamente pelas regras definidas no banco de dados.

2.2 Arquitetura

A arquitetura da solução desenvolvida é composta pela *API metadata* implementada em *Node.js*, a qual se integra com a base de dados MySQL solicitando os metadados das colunas de uma entidade específica. Via *HTTP*, a *API* será consumida pela aplicação do cliente através do *endpoint /metadata/:tableName*, que passa como parâmetro para o método *getMetadata* o nome da entidade a ser consultada e responde com um *JSON* que representa os metadados formatados. A aplicação *web* do cliente precisa incorporar o arquivo ***ValidateField.js***, composto por funções de validação para os campos dos formulários, através dos quais o usuário interage com a aplicação. A organização arquitetural da solução é ilustrada na Figura 1.

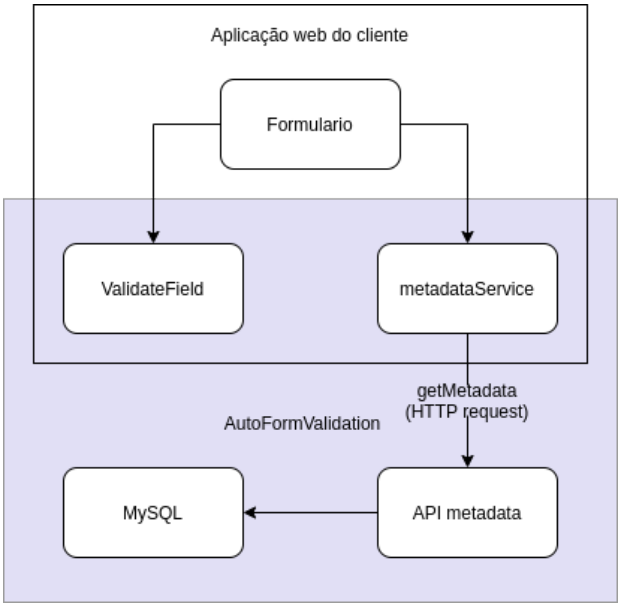

Figura 1: Arquitetura da Solução

2.2.1 Tecnologias

O AutoFormValidation tem suas tecnologias com base em JavaScript, a linguagem mais usada pelos desenvolvedores em 2020, segundo pesquisa do Stack Overflow [8]. Logo, temos a *API metadata* desenvolvida em Node.js, responsável por executar código JS do lado do servidor, e um serviço desenvolvido com React [9], uma biblioteca JS voltada para construção das interfaces de usuário da aplicação. Sobre a base de dados, para ter suporte inicial do AutoFormValidation, foi escolhido o SGBD MySQL por ser um banco de dados relacional e de código aberto, além de ser um dos mais populares [10]. Futuramente, a solução pode vir a suportar outros SGBDs.

2.2.2 Funções de Validação

O *ValidateField.js* é uma solução proposta para verificar o valor inserido pelo usuário em cada campo de entrada de um formulário, usando como base os metadados da entidade equivalente no SGBD da aplicação. O arquivo precisa ser incorporado estaticamente na aplicação *web* do cliente e receber como parâmetro o valor a ser validado e as regras para o mesmo. As funções executam de acordo com o tipo de dado e são isoladas. Isso permite que outras funções adicionais sejam implementadas pelo programador de acordo com as peculiaridades da aplicação *web* do cliente. A Figura 2 apresenta uma dessas funções de validação (*isFieldEmpty*), responsável por verificar se um campo obrigatório foi deixado vazio.

```
const isFieldEmpty = (rule, fieldValue) => {
  const hasError = rule.mandatory && !fieldValue;
  const message = `Field is mandatory`;

  return { hasError, message };
};
```

Figura 2: Exemplo de uma função de validação presente no *ValidateField.js*.

Para implementar uma nova função de validação, como a exemplificada na Figura 2, basta seguir o padrão das demais funções já implementadas. Isso implica que a nova função deve receber *rule* e *fieldValue* como parâmetros, retornar um *object* com as propriedades *hasError* e *message*, e, por fim, associar a função ao tipo equivalente. O parâmetro *rule* é um objeto em que as propriedades são os metadados da coluna correspondente ao campo a ser validado, retornados pela *API metadata*. O parâmetro *fieldValue* é o valor inserido pelo usuário no campo do formulário, e que será verificado conforme a *rule*. Sobre o retorno do método, a propriedade *hasError* é um booleano encarregado de informar a aplicação se o *fieldValue* está de acordo com a *rule*, tendo valor *True* quando alguma regra tiver sido violada. Por fim, a propriedade *message* é um texto breve que descreve, se houver, a violação identificada.

As funções são adicionadas no objeto *validationFnsByType*, apresentado na Figura 3, de acordo com os tipos de dados que são responsáveis por checar. O formulário irá acessar o arquivo somente por meio do método *validateField*, que, com o uso do *validationFnsByType*, define as funções de validação a serem executadas para o campo de entrada. O *validateField* recebe os parâmetros *rule* e *fieldValue* que são passadas para suas funções internas (ex*: isFieldEmpty*), e seu retorno é um *array* composto pelas propriedades *message* de cada função de validação que retornou com valor *True* na propriedade *hasError*.

```
const validationFnsByType = {
  ALL: [isFieldEmpty],
  STRING: [isLengthExceeded],
  NUMBER: [checkUnsigned, checkPrecision, checkScale],
  BOOL: [],
  DATE: [],
};
```

Figura 3: Objeto *validationFnsByType* do método *validateField*, indica quais funções devem ser executadas para cada tipo de dado.

O *ValidateField.js* possui a implementação das seguintes funções:

- *isFieldEmpty*: para validar se um campo obrigatório está vazio;
- *isLengthExceeded*: para checar se o tamanho máximo de caracteres foi excedido;
- *checkUnsigned*: para checar se o valor inserido em um campo numérico deve ser positivo ou não;
- *checkPrecision*: para validar o número máximo de dígitos da parte inteira do valor;
- *checkScale*: para validar o número máximo de casas decimais.

O fluxo do *validateField* consiste em concatenar as funções definidas como valores da chave *ALL* do *validationFnsByType* (que devem ser executadas para todo e qualquer tipo de dado) com as funções inseridas no *validationFnsByType* para o tipo de dado especificado na propriedade *type* do parâmetro *rule*. Com isso, cada função selecionada é executada. Quando uma violação é identificada, o objeto recebido é incorporado no *array* de retorno. A Figura 4 contém o código-fonte descrito.

```
9  const validateField = (rule, fieldValue) => {
10   const errors = [];
11
12 >  const validationFnsByType = {…
18   };
19
20   const validationFns = validationFnsByType['ALL'].concat(
21     validationFnsByType[rule.type]
22   );
23
24   validationFns.forEach((fn) => {
25     const validationResult = fn(rule, fieldValue);
26     if (validationResult.hasError) {
27       errors.push(validationResult.message);
28     }
29   });
30
31   return errors;
32 };
```

Figura 4: Código-fonte da função *validateField*.

2.2.3 *API metadata*

O *metadata*, desenvolvido em Node.js, acessa o SGBD da aplicação *web* cliente usando as credenciais passadas em um arquivo de configuração, obtém os metadados das colunas de uma entidade requisitada e faz todo o processo de limpeza, tratamento e estruturação dos mesmos, tudo isso para que seja retornado um JSON de regras para quem o consome. A Figura 5 apresenta a estrutura de arquivos da API.

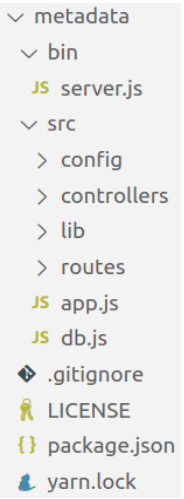

Figura 5: Árvore de diretórios da *API metadata*.

Na raiz do projeto temos como pontos de atenção o *package.json* e as pastas *bin* e *src*. O *package.json* define, entre outras coisas, as dependências e o comando que, chamando o *server.js* da pasta *bin*, rodam a *API metadata*. A pasta *src* concentra o núcleo da API e contém sua estrutura de diretórios conforme ilustrado na Figura 5, onde: *config* contém as credenciais de acesso ao SGBD; *controllers* tem a responsabilidade de obter, tratar e retornar os metadados; *lib* fornece constantes e funções úteis ao controlador da API (*controller)*; enquanto *routes* define o *endpoint* de acesso à API. Além disso, o núcleo é composto pelos arquivos *db.js* e *app.js,* responsáveis pela conexão com o MySQL e a instância da API, respectivamente.

A conexão com o banco de dados é garantida pela definição das credenciais de acesso em *config/dbCredentials.js*. Neste arquivo, é necessário definir o *host*, *database*, *user* e *password* de acesso ao SGBD MySQL. Alternativamente, essas atribuições podem ser feitas via variáveis de ambiente. Esta é a única configuração necessária para executar a API. Feito isso, basta executar o comando *yarn start* na pasta raiz (*metadata*) e a mesma estará rodando.

A propriedade *mandatory* do parâmetro *rule*, presente nas funções de validação, é uma das propriedades retornadas pela API *metadata* para cada coluna de uma entidade específica. O *endpoint /metadata/:tableName* invoca o método *getMetadata* e responde com um JSON composto por um vetor de objetos nos quais as chaves correspondem ao nome das colunas da entidade requisitada e os valores são as regras para validação. A Figura 6 apresenta um exemplo de retorno da API.


```
"name": {
    "defaultValue": null,
    "mandatory": true,
    "type": "STRING",
    "maxLength": 150,
    "precision": null,
    "scale": null,
    "datetime": null,
    "columnType": "varchar(150)",
    "description": ""
},
"weight": {
    "defaultValue": null,
    "mandatory": false,
    "type": "NUMBER",
    "maxLength": null,
    "precision": 5,
    "scale": 2,
    "datetime": null,
    "columnType": "decimal(5,2) unsigned",
    "description": ""
}
```


Figura 6: Trecho de um exemplo do retorno da API *metadata*.

No retorno da *API metadata*, as nove propriedades listadas para os objetos do exemplo mostrado na Figura 6 (colunas *name* e *weight*) são as retornadas para toda e qualquer coluna de uma entidade do SGBD da aplicação *web* do cliente. Cada propriedade possibilita um conjunto de validações que podem ser executadas sobre um campo em um formulário. A seguir, descrevemos o papel de cada uma dessas propriedades:

- *defaultValue*: contém o valor padrão (ou *default*) estabelecido para a respectiva coluna no SGBD). Se não existir, terá valor *null*. Esta informação pode ser usada pela aplicação *web* do cliente para, por exemplo, exibir o formulário com o valor do campo inicialmente preenchido com o valor padrão esperado;
- *mandatory*: indica através de um boolean se o campo é de inserção obrigatória na entidade;
- *type*: contém o tipo JS do dado suportado, baseado no valor da propriedade *columnType* do SGBD. Seus possíveis valores (atualmente suportados) são: STRING, NUMBER, BOOL e DATE;
- *maxLength*: para os campos do *type* STRING, estará preenchido com o valor máximo de caracteres permitidos para a coluna; para os demais, o valor é *null*;
- *precision*: tem valor numérico quando se trata de uma coluna com *type* NUMBER, representando o número máximo de dígitos permitidos para o campo, incluindo as casas decimais, quando presente;
- *scale*: tem valor numérico e representa o número máximo de casas decimais de uma coluna com *type* NUMBER. O valor será *null* para números inteiros ou outros tipos de dados;
- *datetime*: campo para colunas que armazenam dados temporais, ainda não suportadas na versão atual do AutoFormValidation;
- *columnType*: diferente da propriedade *type*, contém o tipo da coluna da entidade conforme o SGBD MySQL. Daqui, é possível extrair algumas informações úteis ao *ValidateField.js*, a exemplo de o campo ser *unsigned* ou não, e determinar a propriedade *type* da API, como é o caso do BOOLEAN atribuído quando o *columnType* é *tinyint(1)*; e
- *description*: descrição da coluna da entidade, se preenchido, pode ser usado pela aplicação *web* do cliente para explicar o que se espera obter como valor do campo.

Estas propriedades apresentadas compõem o objeto *rule* usado pelas funções de validação de campos de formulário do *ValidateField.js*.

A Figura 7 apresenta detalhes do fluxo de execução do *controller* da *API metadata*. Os metadados das colunas de uma entidade do SGBD são obtidos via consultas do tipo SQL SELECT O método *getTableMetadata* é responsável por essa consulta, coletando somente o que é necessário. Ainda no *controller*, o método *formatMetadata* trata os metadados e cria novas propriedades com base em funções pré-estabelecidas, retornando um objeto (*metadata*) que também é o retorno da API.

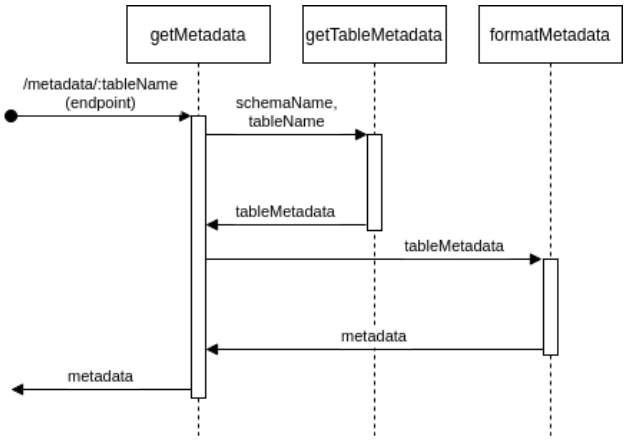

Figura 7: Fluxo do *controller* da API metadata.

2.2.4 *Integração da API com a Aplicação Web do Cliente*

A aplicação *web* cliente que utilizar o AutoFormValidation deverá incorporar o arquivo *ValidateField.js* em seu projeto de desenvolvimento e implementar o consumo da *API metadata* para integrar a solução desenvolvida neste trabalho. Essa integração ocorre no *metadataService* através de uma requisição do tipo HTTP GET, a qual acessa o *endpoint /metadata/:tableName*. Para isso, é necessário passar como parâmetro o nome da entidade da qual será coletada as propriedades. Em seguida, a aplicação *web* do cliente pode invocar o método *validateField* passando as regras de uma das colunas obtidas da API e o valor inserido pelo usuário em um campo do formulário. O retorno é um *array* com as restrições violadas pelo valor do campo em relação às diretrizes da respectiva coluna no SGBD. A seção 3 deste documento apresentará um caso de uso que ajudará a entender o processo de integração do AutoFormValidation com uma aplicação *web* e os benefícios que essa integração pode oferecer.

2.3 Limitações

Considerando a existência de inúmeros SGBDs, principalmente os que contém um grande número de projetos ativos, ter suporte apenas ao MySQL acaba delimitando um grupo de aplicações *web* que podem fazer uso da solução desenvolvida. Este é um fator crítico sobre o qual serão discutidas algumas possíveis soluções de trabalhos futuros na seção 4.3 do presente documento.

Outro ponto relevante é a necessidade de conhecer, no *front-end* da aplicação do cliente, o nome das entidades de banco de dados que terão os campos validados, pois o mesmo é um parâmetro essencial para a *API metadata*. A situação não é tão impactante em casos que o sistema já trabalha com esta informação, de forma dinâmica e automatizada, se comunicando com o *back-end* que possivelmente conhece o nome da entidade, por exemplo. Isso evitaria que o dado fosse inserido estaticamente no código-fonte do *front-end* da aplicação do cliente.

3. CASO DE USO

Nesta seção, vamos analisar a implementação da solução AutoFormValidation por meio de um caso de uso. O objetivo é validar o seu propósito na prática, além de demonstrar o processo de integração com uma aplicação cliente. Primeiramente, para entender a necessidade atendida, será apresentado uma *storytelling* como um cenário de uso. Depois, a solução desenvolvida será aplicada neste cenário. Por último, os resultados são observados e comentados.

3.1 Aplicação de Exemplo

Considere o cenário de uma empresa que precisa validar os campos de formulários desenvolvidos em sua aplicação *web* para receber dados dos usuários. No SGBD MySQL utilizado por essa aplicação, foram definidas regras para uma entidade específica, denominada “employees", onde está definida a obrigatoriedade e o tamanho máximo de 150 caracteres para o campo *name*, ou seja, os metadados da coluna. Se um usuário da aplicação não preencher este campo ou inserir um valor com tamanho superior ao máximo permitido, quando o campo for enviado para ser salvo na base de dados irá gerar uma exceção, pois o mesmo viola as *constraints* pré-estabelecidas.

Daí se tem a necessidade de validar o valor inserido pelo usuário no formulário, para que o mesmo esteja de acordo com os metadados equivalentes. Se esta funcionalidade for implementada estaticamente no código-fonte, ou seja, se houver um trecho de código na aplicação que checa a nulidade do campo *name* e seu número de caracteres diretamente, podemos encontrar situações indesejáveis para o negócio da empresa.

A primeira delas é se, no desenvolvimento, esquecerem de inserir a validação, com a possibilidade de estourar um erro para o usuário final durante a inserção ou edição de dados através de um formulário. A segunda é num cenário em que a validação estática foi implementada na fase de desenvolvimento, mas posteriormente a entidade no SGBD sofreu alterações em suas *constraints*. Isso tornaria a validação implementada inconsistente com a base de dados, podendo até gerar uma exceção na inserção e edição de registros.

Buscando evitar essas situações indesejadas, vamos implementar a solução AutoFormValidation no cenário indicado. Será mostrado como uma validação dinâmica que ocorre através dos metadados da entidade pode não apenas evitar os problemas citados anteriormente, como trazer benefícios no desenvolvimento e manutenção da aplicação, uma vez que não será mais necessário implementar e testar validações estáticas dos campos dos formulários.

Figura 8: Estrutura de arquivos do Caso de Exemplo

Desenvolvido com base em um exemplo de um tutorial [11], o código-fonte da aplicação *web* criada para exemplificar o uso da solução segue a árvore de diretórios apresentada na Figura 8. A seguir, cada nó da árvore *src*, que contém o núcleo da aplicação, é detalhado:

- *components*: contém os componentes React utilizados pela aplicação, subdividido em:
    - *employee*: componentes do CRUD (*Create, Read, Update and Delete*) do sistema;
    - *formFields*: componentes usados pelo formulário para inserção e edição dos campos;
- *constants*: constantes usadas pela aplicação, como por exemplo, representações de valores fixos para o campo *Gênero* do formulário;
- *context*: responsável por carregar o contexto da aplicação, ou seja, trazer informações da base de dados como a lista de funcionários (*employees)*;
- *services*: armazena os serviços, a exemplo do *metadata*, ou *metadataService* da arquitetura da solução AutoFormValidation; e
- *utils*: no qual foi armazenado o *ValidateField.js* do AutoFormValidation.

O arquivo Form.js, em *components/employee*, é o formulário da nossa aplicação de exemplo. É nele onde os metadados obtidos da *API metadata* serão passados para o *ValidateField.js* validar os valores inseridos nos campos. O mesmo, a partir do Add.js ou Edit.js, se responsabiliza pela inserção ou edição de registros da entidade *employees* do SGBD da aplicação e, para isso, faz uso dos componentes criados no *formField*.

O banco de dados da aplicação de exemplo foi criado em um *container* Docker (plataforma *open source* para criação e administração de ambientes isolados) [12]. O arquivo *docker-compose.yml*, necessário para rodar o SGBD, está no diretório *example/database*, ao lado do script de criação da entidade *employees* no SGBD MySQL apresentado na Figura 9.

```sql
CREATE TABLE `employees` (
  `id` int unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(150) NOT NULL,
  `birthday` date NOT NULL,
  `gender` char(1) NOT NULL,
  `height` int unsigned DEFAULT NULL,
  `weight` decimal(5,2) unsigned DEFAULT NULL,
  `has_children` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
)
```

Figura 9: SQL de criação da entidade *employees*.

3.2 Integração do AutoFormValidation

Para implementar o AutoFormValidation, começamos com a criação do *services/metadata.js* (ver Figura 8), que vai implementar o *metadataService* da arquitetura apresentada na seção 2.2. O código-fonte desta implementação é apresentado na Figura 10, onde foi instanciada a URL de acesso à API (linha 3) e definido o método *getMetadata* que acessa o *endpoint* e requisita os metadados da entidade desejada (linhas 5-13).

Em seguida, incorporamos o *ValidateField.js* no diretório *utils*, onde, se necessário, pode ser incrementado com novas funções de validação, a depender da necessidade da aplicação cliente.

```
1  import axios from "axios";
2
3  const METADATA_URL = "http://localhost:8000/metadata/";
4
5  export const getMetadata = (tableName) => {
6    return axios
7      .get(METADATA_URL + tableName)
8      .then((response) => {
9        return response.data;
10     })
11     .catch((error) => {
12       console.log(error);
13     });
14 };
```

Figura 10: Implementação do *metadataService* no *services/metadata.js*.

O último passo da integração é implementar a chamada do *validateField* no código-fonte do formulário, passando o valor inserido e os metadados obtidos com o *getMetadata*. No início do formulário, é instanciada a variável *metadata*, na qual são atribuídos os metadados da entidade relacionada. A Figura 11 apresenta o trecho de código correspondente no formulário. Na linha 20, temos a situação destacada na seção 2.3, em que o nome da entidade do SGBD (ex: *employees*) é necessária para passar como parâmetro da função *getMetadata*.

```
16   const [metadata, setMetadata] = useState({});
17
18   useEffect(() => {
19     async function fetchMetadata() {
20       setMetadata(await getMetadata('employees'));
21     }
22     fetchMetadata();
23   }, []);
```

Figura 11: Instância e atribuição da variável *metadata*.

Ainda nesta etapa, o handleOnChange, método responsável por renderizar cada alteração feita em um campo, é incorporado em uma função nomeada *handleOnChangeWithValidate* (Figura 12). Esta, chama o *validateField*, indica se mensagens relacionadas às regras infringidas devem ser exibidas e, em sequência, altera o valor a ser exibido.

```
60   const handleOnChangeWithValidate = async (field, value) => {
61     const validationErrors
62       = validateField(metadata[field], value);
63
64     validationErrors?.length > 0
65       ? showValidationError(field, validationErrors)
66       : removeValidationError(field);
67
68     handleOnChange(field, value);
69   };
```

Figura 12: Método *handleOnChangeWithValidate* que chama o *validateField*.

Cada componente responsável por um campo do formulário, que deseja validar o valor inserido, deve passar para a propriedade *onChange* do React a função *handleOnChangeWithValidate*, como demonstrado no exemplo apresentado na Figura 13. O processo será o mesmo para todo e qualquer campo, independente do tipo, pois o *ValidateField.js* se encarrega de definir quais funções de validação serão executadas para cada tipo.

```
<Input
  label={'Name'}
  fieldId={'name'}
  onChange={handleOnChangeWithValidate}
  entity={employee}
  inputType={'text'}
></Input>
```

Figura 13: Exemplo de campo do formulário que incorpora a validação na propriedade *onChange*.

Com toda a integração concluída, a solução AutoFormValidation está pronta para funcionar na aplicação de exemplo. Da forma como foi configurada, as violações dos campos do formulário serão apresentadas abaixo dos respectivos campos.

3.3 Resultados Observados

Finalizada a integração, utilizamos um prompt de comando para executar o serviço da *API metadata* com o comando `yarn start` no diretório raiz. Em seguida, iniciamos o *container* Docker com o comando `docker-compose up -d` no diretório *example/database*. Por fim, agora no diretório *crud* da aplicação *web*, executamos o comando `yarn start`.

Com a aplicação de exemplo rodando, temos acesso a uma tela inicial que contém uma listagem de registros da entidade *employees*. Um exemplo desta tela inicial é apresentado na Figura 14.

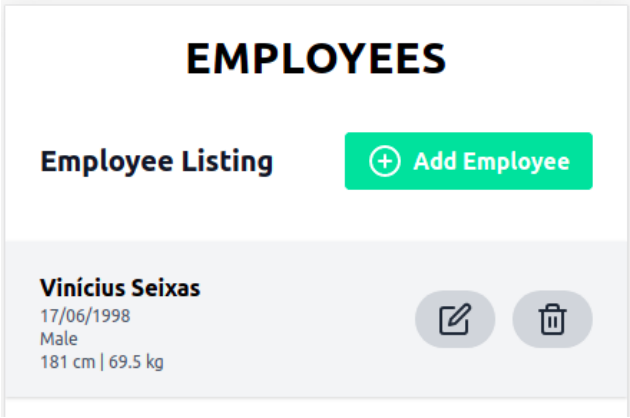

Figura 14: Tela inicial do caso de exemplo.

Para cada item desta lista (ex: Vinícius Seixas, na Figura 14), é possível editar e excluir. A tela também contém um botão para inserção de um novo registro (“Add Employee”). Tanto a ação de inserção quanto a de edição irão redirecionar o usuário da aplicação para a tela de formulário. A Figura 15 mostra o formulário já com algumas validações feitas para os valores inseridos nos campos.

Na Figura 15, podemos ver um exemplo de validação dos campos do formulário da aplicação de exemplo. É possível notar o destaque e os alertas das violações para todos os campos que não estiverem preenchidos de acordo com o esperado pela entidade *employees* do SGBD. Por exemplo: o campo *name* é destacado por ultrapassar o limite máximo de caracteres; *birthday* por não ser preenchido quando obrigatório; *height* e *weight* se destacam por violar regras específicas de campos numéricos, como limites mínimo e máximo.

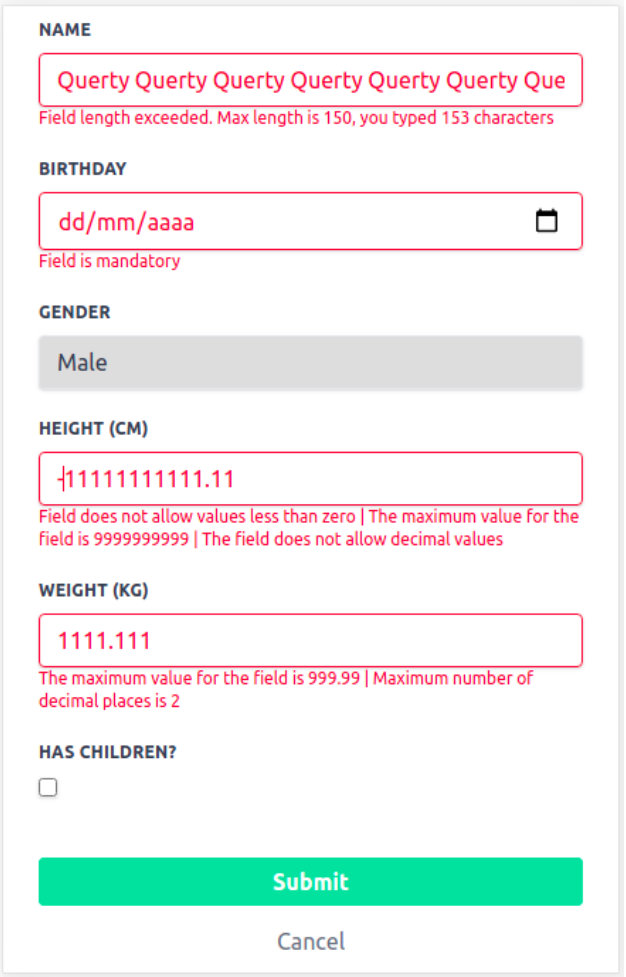

Figura 15: Tela de formulário apresentando o resultado da validação dos valores inseridos nos campos.

Dessa forma, o AutoFormValidation ajudou a impedir que o formulário gere problemas relacionados à quebra de *constraints* do SGBD, além de dinamizar o processo de validação.

4. CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho, foi proposto o AutoFormValidation como solução para o problema de validar, de forma dinâmica, campos de formulários em aplicações *web*. Uma implementação estática para validação pode inserir erros em uma aplicação. Mesmo quando desenvolvida corretamente, a aplicação pode se tornar inconsistente com alterações futuras em *constraints* da entidade equivalente ao formulário na base de dados.

A solução desenvolvida se baseia diretamente nos metadados das colunas da entidade vinculada no SGBD para dinamizar a validação de campos do formulário. A *API metadata* está bem

isolada para facilitar integração e uso, enquanto o *ValidateField.js* se responsabiliza por fornecer as funções de validação.

A *storytelling* estabelecida no caso de uso permitiu analisar a integração e funcionamento do AutoFormValidation na prática. Foi possível visualizar as violações de *constraints* sendo identificadas e notificadas no formulário de exemplo. A partir daí, a aplicação que implementa a solução pode incrementar a funcionalidade conforme suas especificidades.

No processo de concepção e desenvolvimento da solução proposta, foi desafiante definir a arquitetura. Uma possível alternativa para a solução era com as funções de validação na API, o que reduziria o acoplamento gerado pela integração e eliminaria parte da execução de código-fonte no lado do cliente. Porém, a mesma impossibilitaria de validar a entrada no campo do formulário a cada inserção de caractere. A necessidade de diversas requisições à API iria sobrecarregar o servidor da aplicação. Se assim fosse feito, seria viável validar apenas após o envio do formulário. Portanto, manter o *ValidateField.js* validando no lado do cliente foi a solução adotada, o que permitiu uma validação mais reativa.

Embora já possa ser implantado em várias aplicações que utilizem tecnologias *web* em seus formulários, o AutoFormValidation ainda pode ser evoluído em trabalhos futuros. Algumas ideias que podem ser futuramente endereçadas são: fornecer na *API metadata* suporte para diferentes SGBDs, priorizando aqueles mais utilizados [10]; transformar o arquivo *ValidateField.js* em uma biblioteca JS, o que permitiria a injeção por dependência e maior facilidade na integração com a aplicação usuária da solução; e, por fim, implementar novas funções de validação, expandindo os cenários já considerados.

5. AGRADECIMENTOS

Agradeço aos meus pais (Valdevir e Eliane) e irmãos (Victor e Gabriela) por sempre me apoiarem. Agradeço à minha namorada Isabelle por todo amor, suporte e companheirismo. Agradeço ao meu amigo Francisco por todo auxílio prestado. Agradeço aos meus demais amigos presentes em minha jornada, Erilânia, Ana Beatriz, Ignácio, Emerson, Gustavo e Raoni. Por fim, agradeço ao meu orientador, Prof. Maxwell, pela sabedoria, disponibilidade e paciência ao me acolher no desenvolvimento deste trabalho.

6. REFERÊNCIAS

[1] NS4B - NET SOLUTION FOR BUSINESS LTDA. “Sistemas de informação”, 9 de junho de 2016, http://ns4business.com.br/sistemas-de-informacao/. Acesso em 8 de maio de 2021.

[2] Abraham Silberschatz, Henry F. Korth, S. Sudarshan., Sistema de Banco de Dados, 3 Edição, Makron Books, 2000.

[3] Elmasri, R. e Navathe, S. “3.2 Restrições em modelo relacional e esquemas de bancos de dados relacionais.” Sistemas de banco de dados, 6a ed., Pearson Education, 2011, p, 44-45.

[4] Guerra, Bruno. “O que é integridade de dados e por que ela é importante?” Inteligência de Negócios, 6 de janeiro de 2020, https://blog.in1.com.br/o-que-%C3%A9-integridade-de-dados-e-por-que-ela-%C3%A9-importante. Acesso em 8 de maio de 2021.

[5] OpenJS Foundation. Node.js, https://nodejs.org/. Acesso em 11 de maio de 2021.

[6] Oracle Corporation and/or its affiliates. MySQL, https://www.mysql.com/. Acesso em 11 de maio de 2021.

[7] Oracle Corporation. “The INFORMATION_SCHEMA COLUMNS Table.” MySQL 8.0 Reference Manual, 1995, https://dev.mysql.com/doc/refman/8.0/en/information-schema-columns-table.html. Acesso em 30 de março de 2021.

[8] Stack Overflow. “Most Popular Technologies.” Stack Overflow Developer Survey 2020, 2020, https://insights.stackoverflow.com/survey/2020#most-popular-technologies. Acesso em 30 de março de 2021.

[9] Facebook Inc. React, https://reactjs.org/. Acesso em 12 de maio de 2021.

[10] DB-Engines. “DB-Engines Ranking.” DB-Engines Ranking, 2021, https://db-engines.com/en/ranking. Acesso em 30 de março de 2021.

[11] Ishan Manandhar. “How To Build a CRUD App with React Hooks and the Context API.” DigitalOcean Community, 16 de março de 2020, https://www.digitalocean.com/community/tutorials/react-crud-context-hooks. Acesso em 1 de maio de 2021.

[12] Docker Inc. “Docker Overview.” Docker Docs, 2013-2021, https://docs.docker.com/get-started/overview/. Acesso em 1 de maio de 2021.