Validação de Formulários

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

Validação de formulários é onde a borracha encontra o asfalto na implementação de autenticação. É aqui que transformamos especificações abstratas de segurança em código concreto que guia usuários a fornecer dados corretos antes mesmo de tentar enviar ao servidor. Uma validação bem implementada não apenas previne erros, mas melhora a experiência ao dar feedback imediato e claro sobre o que precisa ser corrigido.

Nesta seção, vamos construir um sistema completo de validação que é robusto o suficiente para proteger contra dados inválidos, mas amigável o suficiente para não frustrar usuários. Vamos aprender a validar cada tipo de campo que aparece em telas de autenticação, desde emails até senhas complexas, usando as ferramentas nativas do Flutter de forma eficaz.

Validação de Email: Mais Complexa do Que Parece

Validar um endereço de email parece simples à primeira vista, mas há nuances importantes que fazem a diferença entre uma validação que funciona bem e uma que causa problemas.

O Desafio da Validação de Email

O formato técnico oficial de emails é surpreendentemente complexo, definido por várias RFCs que permitem construções que raramente vemos na prática. Emails podem tecnicamente incluir caracteres especiais em lugares inesperados, ter múltiplos níveis de subdomínios, e até incluir espaços quando properly quoted. Implementar validação que aceita todos os formatos tecnicamente válidos seria extremamente complexo e provavelmente desnecessário para a maioria das aplicações.

A abordagem pragmática é validar contra o formato que noventa e nove por cento dos emails reais usam, enquanto aceita que casos extremamente raros podem ser rejeitados. Para aplicações de autenticação, queremos verificar que o email tem a estrutura básica: algo antes do arroba, o arroba em si, um domínio, e uma extensão de domínio com pelo menos dois caracteres.

Implementando Validação Prática de Email

Aqui está uma função validadora que equilibra rigor com praticidade:

dart

```dart
class Validators {
  /// Valida formato de email usando regex que aceita formatos comuns
  /// mas não necessariamente todos os formatos tecnicamente válidos
  static String? validateEmail(String? value) {
    // Primeiro, verifica se campo está vazio
    if (value == null || value.trim().isEmpty) {
      return 'Por favor, informe seu email';
    }

    // Remove espaços em branco no início e fim
    final email = value.trim();

    // Regex que valida formato básico de email
    // Aceita: letras, números, pontos, hífens, underscores antes do @
    // Exige @ seguido de domínio válido com ponto e extensão
    final emailRegex = RegExp(
      r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    );

    if (!emailRegex.hasMatch(email)) {
      return 'Por favor, informe um email válido';
    }

    // Verificação adicional: domínio não pode começar ou terminar com ponto
    final parts = email.split('@');
    if (parts.length != 2) {
      return 'Por favor, informe um email válido';
    }

    final domain = parts[1];
    if (domain.startsWith('.') || domain.endsWith('.')) {
      return 'Por favor, informe um email válido';
    }

    // Tudo certo, retorna null indicando validação passou
    return null;
  }
}
```

Esta implementação cobre os casos mais comuns que você encontrará. Ela aceita endereços padrão como usuario@exemplo.com, aceita subdomínios como usuario@mail.exemplo.com, aceita caracteres especiais comuns no nome local como usuario.nome@exemplo.com ou usuario+filtro@exemplo.com, e rejeita formatos claramente inválidos como @exemplo.com ou usuario@.com.

O ponto importante é que retornamos null quando a validação passa, não true ou uma string vazia. O Flutter interpreta null como sucesso na validação. Qualquer string retornada é tratada como mensagem de erro a ser

exibida ao usuário.

Normalização de Email

Antes de validar, normalizamos o email removendo espaços em branco no início e fim com trim. Isto é importante porque usuários frequentemente copiam e colam emails de outras fontes e podem inadvertidamente incluir espaços extras. Sem normalização, um email válido seria rejeitado apenas por ter um espaço ao final, frustrando o usuário.

Considere também converter o email para minúsculas antes de enviá-lo ao servidor, já que emails são case-insensitive na parte do domínio e geralmente também na parte local. Isto previne problemas onde Usuario@exemplo.com e usuario@exemplo.com são tratados como contas diferentes:

```dart
final normalizedEmail = email.trim().toLowerCase();
```

Aplique esta normalização tanto na validação quanto ao enviar para o servidor, garantindo consistência.

Validação de Senha: Equilibrando Segurança e Usabilidade

Senhas apresentam um desafio interessante porque queremos que sejam fortes o suficiente para resistir a ataques, mas não tão complexas que usuários não consigam lembrar ou fiquem frustrados tentando criar uma que seja aceita.

Definindo Requisitos Razoáveis

A tentação ao definir requisitos de senha é ser extremamente rigoroso, exigindo letras maiúsculas, minúsculas, números, caracteres especiais, comprimento mínimo de doze caracteres, e assim por diante. Embora isto tecnicamente crie senhas mais fortes, também cria fricção significativa que pode afastar usuários.

Pesquisas recentes em segurança sugerem que comprimento é mais importante que complexidade para força de senha. Uma senha de doze caracteres contendo apenas letras minúsculas é matematicamente mais forte que uma senha de oito caracteres com todos os tipos de caracteres. Além disso, requisitos muito rígidos levam usuários a criar senhas previsíveis que seguem padrões para atender os requisitos, como Senha123! ou Password1@.

Uma política equilibrada para a maioria das aplicações é exigir mínimo de oito caracteres com pelo menos uma letra e um número. Isto é forte o suficiente para resistir a ataques de força bruta básicos enquanto permanece acessível para usuários:

```dart
```

```dart
class Validators {
  /// Valida senha com requisitos balanceados de segurança
  static String? validatePassword(String? value) {
    if (value == null || value.isEmpty) {
      return 'Por favor, informe uma senha';
    }

    // Verifica comprimento mínimo
    if (value.length < 8) {
      return 'A senha deve ter no mínimo 8 caracteres';
    }

    // Verifica presença de pelo menos uma letra
    if (!value.contains(RegExp(r'[a-zA-Z]'))) {
      return 'A senha deve conter pelo menos uma letra';
    }

    // Verifica presença de pelo menos um número
    if (!value.contains(RegExp(r'[0-9]'))) {
      return 'A senha deve conter pelo menos um número';
    }

    return null;
  }
}
```

Validação de Força em Tempo Real

Ao invés de apenas rejeitar senhas fracas, podemos educar o usuário mostrando a força da senha que ele está criando. Isto transforma validação em orientação proativa:

dart

```dart
class PasswordStrength {
  /// Calcula força da senha em escala de 0.0 (fraquíssima) a 1.0 (forte)
  static double calculate(String password) {
    if (password.isEmpty) return 0.0;

    double score = 0.0;

    // Pontuação por comprimento (até 40% do total)
    if (password.length >= 8) score += 0.2;
    if (password.length >= 12) score += 0.1;
    if (password.length >= 16) score += 0.1;

    // Pontuação por variedade de caracteres (60% do total)
    if (password.contains(RegExp(r'[a-z]'))) score += 0.15;
    if (password.contains(RegExp(r'[A-Z]'))) score += 0.15;
    if (password.contains(RegExp(r'[0-9]'))) score += 0.15;
    if (password.contains(RegExp(r'[!@#$%^&*(),.?":{}|<>]'))) score += 0.15;

    return score.clamp(0.0, 1.0);
  }

  /// Retorna descrição textual da força
  static String getStrengthText(double strength) {
    if (strength < 0.3) return 'Muito fraca';
    if (strength < 0.5) return 'Fraca';
    if (strength < 0.7) return 'Média';
    if (strength < 0.9) return 'Forte';
    return 'Muito forte';
  }

  /// Retorna cor apropriada para o nível de força
  static Color getStrengthColor(double strength) {
    if (strength < 0.3) return Colors.red;
    if (strength < 0.5) return Colors.orange;
    if (strength < 0.7) return Colors.yellow.shade700;
    return Colors.green;
  }
}
```

Esta classe fornece métodos utilitários que você pode usar em widgets para mostrar feedback visual sobre a força da senha. A pontuação considera tanto comprimento quanto variedade de caracteres, dando crédito incremental por cada melhoria. Um usuário pode ver sua senha passar de vermelha para amarela para verde à medida que a fortalece, criando um feedback loop positivo.

Verificação de Senhas Comuns

Uma camada adicional de segurança que aplicações sérias implementam é verificar se a senha escolhida está em listas de senhas comumente usadas. Senhas como password, 123456, ou qwerty aparecem em praticamente todos os ataques de dicionário e nunca deveriam ser aceitas.

Você pode manter uma lista local das mil senhas mais comuns e verificar contra ela:

```dart
class Validators {
  // Lista das senhas mais comuns que nunca devem ser aceitas
  static const _commonPasswords = {
    'password', '123456', '12345678', 'qwerty', 'abc123',
    'monkey', '1234567', 'letmein', 'trustno1', 'dragon',
    // ... adicione mais conforme necessário
  };

  static String? validatePasswordNotCommon(String? value) {
    if (value == null) return null;

    final lowercase = value.toLowerCase();

    if (_commonPasswords.contains(lowercase)) {
      return 'Esta senha é muito comum. Por favor, escolha outra.';
    }

    return null;
  }
}
```

Validação de Confirmação de Senha

A confirmação de senha existe para capturar erros de digitação antes que o usuário complete o cadastro e descubra que não consegue fazer login porque digitou a senha errada. A validação é conceitualmente simples mas há detalhes de implementação importantes.

Comparação Direta

A validação de confirmação compara diretamente com o valor do campo de senha original. Para fazer isto, a função validadora precisa ter acesso ao valor do campo de senha:

```dart
```

```dart
class RegisterForm extends StatefulWidget {
  @override
  _RegisterFormState createState() => _RegisterFormState();
}

class _RegisterFormState extends State<RegisterForm> {
  final _formKey = GlobalKey<FormState>();
  final _passwordController = TextEditingController();
  final _confirmPasswordController = TextEditingController();

  @override
  void dispose() {
    _passwordController.dispose();
    _confirmPasswordController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        children: [
          TextFormField(
            controller: _passwordController,
            obscureText: true,
            decoration: InputDecoration(labelText: 'Senha'),
            validator: Validators.validatePassword,
          ),
          SizedBox(height: 16),
          TextFormField(
            controller: _confirmPasswordController,
            obscureText: true,
            decoration: InputDecoration(labelText: 'Confirmar senha'),
            validator: (value) => _validatePasswordConfirmation(value),
          ),
        ],
      ),
    );
  }

  String? _validatePasswordConfirmation(String? value) {
    if (value == null || value.isEmpty) {
      return 'Por favor, confirme sua senha';
    }
```

```
    // Compara diretamente com o valor do campo de senha
    if (value != _passwordController.text) {
      return 'As senhas não coincidem';
    }

    return null;
  }
}
```

Note como a função de validação tem acesso ao passwordController, permitindo comparar os valores diretamente. Este padrão funciona bem quando a validação está definida inline no mesmo StatefulWidget que mantém os controllers.

Revalidação Quando Senha Muda

Um detalhe importante é que se o usuário preenche ambos os campos corretamente, mas depois volta e muda apenas o campo de senha, o campo de confirmação pode não mostrar erro imediatamente porque ele não foi modificado. Para lidar com isto, escute mudanças no campo de senha e force revalidação do campo de confirmação:

dart

```
@override
void initState() {
  super.initState();

  // Quando senha muda, revalida confirmação se ela já foi preenchida
  _passwordController.addListener(() {
    if (_confirmPasswordController.text.isNotEmpty) {
      // Força revalidação do form para atualizar campo de confirmação
      _formKey.currentState?.validate();
    }
  });
}
```

Isto garante que se os campos ficarem dessincronizados, o erro aparece imediatamente no campo de confirmação, não apenas quando o usuário tenta submeter o form.

Validação de Campos Obrigatórios

Para campos que devem ser preenchidos mas não têm requisitos especiais de formato, a validação verifica apenas que não estão vazios:

dart

```dart
class Validators {
  /// Validação genérica para qualquer campo obrigatório
  static String? validateRequired(String? value, String fieldName) {
    if (value == null || value.trim().isEmpty) {
      return 'Por favor, informe $fieldName';
    }
    return null;
  }

  /// Validação específica para nome
  static String? validateName(String? value) {
    if (value == null || value.trim().isEmpty) {
      return 'Por favor, informe seu nome';
    }

    // Opcional: verifica comprimento mínimo
    if (value.trim().length < 2) {
      return 'Nome deve ter pelo menos 2 caracteres';
    }

    return null;
  }
}
```

O uso de trim é importante porque previne que usuários "preencham" o campo apenas com espaços. Um campo contendo apenas espaços em branco é tratado como vazio.

A versão genérica validateRequired aceita o nome do campo como parâmetro, permitindo reutilizar a mesma função para diferentes campos com mensagens apropriadas. Você pode usá-la assim:

```dart
TextFormField(
  decoration: InputDecoration(labelText: 'Nome completo'),
  validator: (value) => Validators.validateRequired(value, 'seu nome'),
)
```

Implementando Validação com Form e GlobalKey

O Flutter fornece widgets especializados que tornam validação de formulários limpa e eficiente. O widget Form atua como container para campos de entrada, e GlobalKey permite acessar o estado do Form para validar todos os campos de uma vez.

Estrutura Básica de um Form

A estrutura fundamental de um formulário validável no Flutter envolve três componentes principais trabalhando juntos:

dart

```dart
class LoginPage extends StatefulWidget {
  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  // GlobalKey permite acessar o estado do Form
  final _formKey = GlobalKey<FormState>();

  // Controllers para acessar valores dos campos
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();

  bool _isLoading = false;

  @override
  void dispose() {
    // Sempre limpar controllers quando não precisar mais deles
    _emailController.dispose();
    _passwordController.dispose();
    super.dispose();
  }

  Future<void> _handleSubmit() async {
    // Valida todos os campos do formulário
    if (_formKey.currentState!.validate()) {
      // Todos os campos são válidos, prosseguir com login
      setState(() => _isLoading = true);

      try {
        await authService.login(
          _emailController.text.trim().toLowerCase(),
          _passwordController.text,
        );

        // Login bem-sucedido, navegar para home
        Navigator.of(context).pushReplacementNamed('/home');

      } catch (e) {
        // Login falhou, mostrar erro
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text('Erro ao fazer login: $e')),
        );
      } finally {
        if (mounted) {
          setState(() => _isLoading = false);
```

```
      }
    }
  } else {
    // Validação falhou, erros já estão sendo mostrados nos campos
    // Opcionalmente, fazer scroll para o primeiro erro
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Login')),
    body: SingleChildScrollView(
      padding: EdgeInsets.all(24),
      child: Form(
        key: _formKey, // Associa GlobalKey ao Form
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [
            TextFormField(
              controller: _emailController,
              keyboardType: TextInputType.emailAddress,
              textInputAction: TextInputAction.next,
              decoration: InputDecoration(
                labelText: 'Email',
                prefixIcon: Icon(Icons.email_outlined),
              ),
              validator: Validators.validateEmail,
            ),
            SizedBox(height: 16),
            TextFormField(
              controller: _passwordController,
              obscureText: true,
              textInputAction: TextInputAction.done,
              decoration: InputDecoration(
                labelText: 'Senha',
                prefixIcon: Icon(Icons.lock_outlined),
              ),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Por favor, informe sua senha';
                }
                return null;
              },
              onFieldSubmitted: (_) => _handleSubmit(),
            ),
            SizedBox(height: 24),
```

```
            ElevatedButton(
              onPressed: _isLoading ? null : _handleSubmit,
              child: _isLoading
                ? CircularProgressIndicator()
                : Text('Entrar'),
            ),
          ],
        ),
      ),
    ),
  );
 }
}
```

Como a Validação Funciona Internamente

Quando você chama formKey.currentState.validate, o Flutter percorre todos os TextFormFields dentro do Form e executa suas funções validator. Se qualquer validator retornar uma string não-nula, essa string é exibida como erro abaixo do campo correspondente e o método validate retorna false. Se todos os validators retornarem null, o método retorna true indicando que o formulário é válido.

Esta abordagem declarativa é elegante porque cada campo conhece suas próprias regras de validação através da função validator, e o Form coordena a validação de todos eles com uma única chamada de método. Você não precisa escrever código imperativo checando cada campo manualmente.

AutovalidateMode para Validação Progressiva

Por padrão, a validação só acontece quando você chama explicitamente validate. Porém, você pode configurar o Form para validar automaticamente em diferentes momentos usando autovalidateMode:

```dart
Form(
  key: _formKey,
  autovalidateMode: AutovalidateMode.onUserInteraction,
  child: // ... campos
)
```

As opções são disabled que só valida quando você chama validate explicitamente, onUserInteraction que valida quando o usuário interage com qualquer campo, e always que valida constantemente. Geralmente onUserInteraction oferece o melhor equilíbrio, validando após o usuário começar a interagir mas não antes.

Validação Assíncrona

Algumas validações requerem comunicação com o servidor, como verificar se um email já está cadastrado ou se

um nome de usuário está disponível. Estas validações assíncronas têm considerações especiais.

O Desafio da Validação Assíncrona

O problema com validação assíncrona é que ela pode levar tempo variável dependendo da latência da rede. Se você valida a cada tecla pressionada, pode sobrecarregar o servidor com centenas de requisições e criar uma experiência confusa onde o feedback está sempre atrasado em relação à digitação.

A solução é usar debouncing para esperar que o usuário pause na digitação antes de fazer a verificação, e mostrar um indicador de loading enquanto a verificação acontece:

dart

```dart
class _RegisterPageState extends State<RegisterPage> {
  Timer? _debounceTimer;
  bool _isCheckingEmail = false;
  bool? _emailAvailable;

  void _onEmailChanged(String value) {
    // Cancela timer anterior se existe
    _debounceTimer?.cancel();

    // Limpa estado de disponibilidade enquanto usuário digita
    setState(() {
      _emailAvailable = null;
    });

    // Só faz verificação se o email tem formato válido
    if (Validators.validateEmail(value) != null) {
      return;
    }

    // Cria novo timer que dispara após 500ms de inatividade
    _debounceTimer = Timer(Duration(milliseconds: 500), () {
      _checkEmailAvailability(value);
    });
  }

  Future<void> _checkEmailAvailability(String email) async {
    setState(() => _isCheckingEmail = true);

    try {
      final available = await authService.checkEmailAvailable(email);

      if (mounted) {
        setState(() {
          _emailAvailable = available;
          _isCheckingEmail = false;
        });
      }
    } catch (e) {
      if (mounted) {
        setState(() => _isCheckingEmail = false);
      }
    }
  }

  @override
  void dispose() {
```

```
    _debounceTimer?.cancel();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return TextFormField(
      decoration: InputDecoration(
        labelText: 'Email',
        suffixIcon: _isCheckingEmail
            ? SizedBox(
                width: 20,
                height: 20,
                child: Padding(
                  padding: EdgeInsets.all(12),
                  child: CircularProgressIndicator(strokeWidth: 2),
                ),
              )
            : _emailAvailable != null
                ? Icon(
                    _emailAvailable! ? Icons.check_circle : Icons.error,
                    color: _emailAvailable! ? Colors.green : Colors.red,
                  )
                : null,
      ),
      validator: (value) {
        final basicValidation = Validators.validateEmail(value);
        if (basicValidation != null) return basicValidation;

        // Validação assíncrona só é checada no momento do submit
        if (_emailAvailable == false) {
          return 'Este email já está cadastrado';
        }

        return null;
      },
      onChanged: _onEmailChanged,
    );
  }
}
```

Este padrão mostra três estados visuais claros: loading enquanto verifica, check verde se disponível, erro vermelho se já existe. O debouncing garante que só fazemos uma verificação quando o usuário realmente parou de digitar, não a cada caractere.

Composição de Validadores

Quando um campo precisa passar por múltiplas validações, você pode compor validators para manter o código organizado:

```dart
class Validators {
  /// Compõe múltiplos validators em um único
  /// Retorna o primeiro erro encontrado, ou null se todos passarem
  static String? Function(String?) compose(
    List<String? Function(String?)> validators,
  ) {
    return (String? value) {
      for (final validator in validators) {
        final error = validator(value);
        if (error != null) return error;
      }
      return null;
    };
  }
}

// Uso:
TextFormField(
  validator: Validators.compose([
    Validators.validateRequired,
    Validators.validateEmail,
    (value) => _emailAvailable == false
      ? 'Email já cadastrado'
      : null,
  ]),
)
```

Esta abordagem mantém cada validação focada em uma responsabilidade específica enquanto permite combiná-las facilmente.

Conclusão da Seção

Nesta seção, construímos um sistema completo de validação de formulários que é robusto, amigável ao usuário, e maintível. Aprendemos a validar emails com pragmatismo, criar requisitos de senha balanceados, implementar confirmação de senha com revalidação automática, usar o sistema nativo de Forms do Flutter efetivamente, e lidar com validações assíncronas usando debouncing.

Validação eficaz é invisível quando funciona bem. Usuários não pensam sobre ela conscientemente, apenas notam que o aplicativo os guia suavemente a fornecer dados corretos. Esta é a marca de validação bem implementada: ela previne erros sem criar fricção.

Na próxima seção, vamos explorar como gerenciar o estado de autenticação em toda a aplicação usando Riverpod, permitindo que múltiplas telas reajam automaticamente quando o usuário faz login ou logout.

Fim da Seção 6: Validação de Formulários