**Branches e integração contínua: o problema de feature branches**

Integrar o código criado pelos desenvolvedores o mais frequentemente possível, com espaços de tempo mínimos, para que o feedback e consequências do código criado por um desenvolvedor entre em contato o mais rapidamente possível com os outros é o processo chamado de integração contínua.

Mas em diversos momentos existe a tentação de criar branches, linhas separadas de desenvolvimento, para cada funcionalidade: um feature branch, em cada um deles um conjunto de desenvolvedores trabalha de maneira isolada. [O problema de desenvolver muito tempo sem integrar é que no momento que um feature branch terminar integrando-se ao mainline, os outros branches terão um processo de merge grande e custoso a ser feito](http://martinfowler.com/bliki/FeatureBranch.html).

Variações do "feature branch" são branches por cliente, onde cada um possui um conjunto distinto de funções que está disponível. Gerenciar esse trabalho é difícil e propenso a erro como por exemplo quando bugs que são corrigidos em determinados branches não são passados para outros. Para mantermos os benefícios da integração contínua, tentamos minimizar o impacto dos feature branches e encontrar uma maneira de remover o efeito negativo dos mesmos, mantendo os benefícios.

Se o branch vive menos de um dia, por exemplo, a prática de feature branches não vira uma discussão: o merge e seu respectivo impacto é pequeno, não é necessário se preocupar com o impacto dessa decisão na integração contínua pois o tempo no qual a integração não é contínua é relativamente baixo.

Por outro lado, existem dois possíveis motivos para um branch viver mais de um dia. O primeiro deles é o de criar uma história muito grande, que para ser implementada é necessário muitas horas de trabalho. A solução aqui é simples: quebrar as histórias em pedaços menores permitirá que os desenvolvedores terminem mais cedo algo entregável que possivelmente resultará em feedback para os próximos trechos a serem criados.

O segundo motivo, mais complicado, é para manter um branch master (ou trunk) limpo que pode ser deployado para produção a qualquer instante. O motivo é justo e nobre uma vez que boas práticas como integração contínua andam de mãos dadas com entrega contínua e feedback frequente mas ao invés de criarmos um bloqueio para algo melhor (a integração contínua), é mais interessante ganhar o benefício de ambos.

Durante o tempo de [compilação](https://www.alura.com.br/artigos/o-que-e-compilacao), build do artefato de deploy ou ainda execução podemos utilizar técnicas para não registrar componentes de funcionalidades ainda não prontos. O exemplo a seguir mostra como utilizar um container simples de injeção de dependências para controlar diversos ambientes com features distintas. Todos os métodos que retornam objetos do tipo Ambiente são utilizados por convenção:

public class Ambientes { public Ambiente desenvolvimento() { return homologacao().com(OpenIdLogin.class); }

public Ambiente homologacao() { return producao().com(ExcelExport.class); }

public Ambiente producao() { return new Ambiente(new Container()).com(ClienteController.class).e(ProjetoController.class).e(LoginController.class); } }

Da mesma maneira, em aplicações web, arquivos jsp templates de qualquer template engine podem ser customizados para mostrar determinadas partes de acordo com o ambiente atual:

<feature name="OpenId"> ... codigo html ... </feature>

E em qualquer ponto do sistema é possível verificar a disponibilidade de uma feature:

public class LoginController { private final FeatureSet features; public LoginController(FeatureSet features) { this.features = features; }

public Usuario login(Map parametros) { LoginFeature login = features.for(Login.class); return login.valida(...); } }

[Toggles limpos e bem elaborados como esses](http://blog.jayfields.com/2010/10/experience-report-feature-toggle-over.html) permitem total customização do seu sistema sem ser um peso tão grande no desenvolvimento. Outras opções de implementações menos rebuscadas mas tão eficientes quanto envolvem a simples leitura de arquivos de propriedades. [Ferramentas como o envie](https://github.com/caelum/envie) adicionam suporte a feature toggles em qualquer ponto de uma aplicação. O mesmo container acima funciona em Ruby da seguinte maneira:

production = Envie.production.with(:oauth) development = production.derive.with(:openid)

feature(:oauth) do *# codigo que utiliza oauth end*

feature(:openid) do *# codigo que utiliza openid end*

Utilizando tais ferramentas de toggle, que são baseadas na criação de enviroments, é possível automatizar o processo de oficialização de uma feature. Dada que a feature "openid" foi aceita, um script é capaz de adiciona-la no ambiente de produção, ou uma variação que removeria as definições de chamada ao métod feature, resultando em um código mais limpo a partir do próximo commit.

Feature branches e toggles são uma alternativa extremamente popular no desenvolvimento open source e muito bem feita em alguns projetos como o kernel do linux que pode ser construído utilizando inúmeras combinações. Testes de integração não são mais capazes de garantir o comportamento através de todas as possíveis combinações mas continuam adicionando valor.

[Durante o www 2010, Ian Robinson](http://iansrobinson.com/) descreveu como o Neo4j pode ser utilizado como alternativa para propragação da configuração de nós diferentes em clusters de máquinas de determinadas aplicações de larga escala na web. A abordagem de feature toggles como esses é interessante e positiva para projetos grandes que envolvem a constante necessidade de integrar funcionalidades ainda não aprovadas por completo e serve como segunda opção durante a utilização de feature branches. Também serve como solução simples para feature toggle por grupos de clientes.

A primeira regra no instante de criar um feature branch é: evite criá-lo. A segunda regra é, integre-o ao mainline o mais rápido possível.

*"Integração Contínua é uma pratica de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente - podendo haver multiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente."*[*Martin Fowler*](http://martinfowler.com/articles/continuousIntegration.html)

Integração Contínua tornou-se muito importante na comunidade de desenvolvimento de software e isso provavelmente ocorreu devido ao grande impacto causado pelas metodologias ágeis. Em equipes que adotaram tais metodologias (eXtreme Programming, Scrum, entre outras), integração contínua é um dos pilares da agilidade, garantindo que todo o sistema funcione a cada build de forma coesa, mesmo que sua equipe seja grande e diversas partes do código estejam sendo alteradas ao mesmo tempo.

Mas porque fazer Integração Contínua ? Quais os benefícios que isso pode trazer ?

Basicamente, a grande vantagem da integração contínua está no **feedback instantâneo**. Isso funciona da seguinte forma: a cada commit no repositório, o build é feito automáticamente, com todos os testes sendo executados de forma automática e falhas sendo detectadas. Se algum commit não compilar ou quebrar qualquer um dos testes, a equipe toma conhecimento instantâneamente (através de email, por exemplo, indicando as falhas e o commit causador das mesmas). A equipe pode então corrigir o problema o mais rápido possível, o que é fundamental para não introduzir erros ao criar novas funcionalidades, refatorar, etc. Integração contínua é mais uma forma de trazer **segurança em relação a mudanças**: você pode fazer modificações sem medo, pois será avisado caso algo saia do esperado.

Mas porque eu não rodo pessoalmente os testes na minha máquina e só então faço o commit? Simples: seu projeto pode ser tão grande, que os testes (em especial os de aceite) demoram um tempo considerável para serem executados e você não vai querer esperar todo esse tempo a cada commit pra poder continuar a trabalhar. Nesse caso, o recomendado é rodar os testes que envolvem as partes que você modificou e só então commitar, deixando para o **servidor de integração contínua** o trabalho de realizar todos os testes do sistema e garantir que tudo esteja funcionando. Além disso, não estamos falando apenas de testes, estamos falando de builds completos: a cada commit temos uma versão que teoricamente está pronta para entrar em produção, e isso pode envolver a realização de tarefas que não faríamos se estivessemos só testando, como por exemplo gerar um arquivo .war. O projeto pode ainda ser implantado automaticamente num servidor de desenvolvimento/homologação, e então com isso a cada commit temos o projeto rodando na web instantaneamente refletindo nossas mudanças!

Um outro exemplo interessante é o processo de geração das apostilas dos nossos cursos: a cada commit feito no repositório, o servidor faz o check-out, executa o **[Tubaina](https://github.com/caelum/tubaina" \o "Tubaína" \t "_blank)**, que transforma o fonte em latex, convertendo o latex gerado em pdf e por fim copiando esse pdf num diretório interno, pronto para impressão na gráfica. Dessa maneira, garantimos que disponibilizamos para nossos alunos o material mais atualizado disponível. Para tudo isso funcionar, no entanto, é preciso agarrar a idéia de commits pequenos. Fica mais fácil saber onde foi introduzido um erro quando o build quebrar se houveram pequenas mudanças, do que ter de verificar as ultimas 50 classes alteradas no ultimo commit. Outro ponto importante, é garantir ao menos um build limpo, com todos os testes passando, ao final de cada dia. Assim, teremos software pronto para entrar em produção tão cedo seja necessário.

Em outras palavras, podemos descrever Integração Contínua como integração **automática** com processo de build **automático** e que roda testes de forma **automática** e **automaticamente** detecta falhas em cada pedaço.

Abaixo as ferramentas que usamos aqui na Caelum:

* [**CruiseControl.rb**](http://cruisecontrolrb.thoughtworks.com/): desenvolvida pela [ThoughtWorks](http://www.thoughtworks.com/" \t "_blank), é a aplicação de integração contínua. Capaz de constantemente verificar os repositórios em busca de novos commits, fazendo check-out e rodando tarefas pré-determinadas. O interessante dessa ferramenta é que ela trabalha com qualquer tipo de projeto: ruby, java, ou qualquer outro cujo build possa ser feito através da linha de comando. Além disso, tem sua interface extremamente simples e funcional :).

![CruiseControl.rb](assets/integracao-continua/cruisecontrol-300x229.png "CruiseControl.rb")

![CruiseControl.rb build report](assets/integracao-continua/cruise-stella-300x196.png "CruiseControl.rb build report")

* [**CCMenu**](http://ccmenu.sourceforge.net/), ou **[CCTray](http://ccnet.sourceforge.net/CCNET/CCTray.html" \t "_blank)**: permite acompanhar o build de cada projeto sem ter de entrar no site do CruiseControl.rb. Isso é feito através de um ícone alertando quais projetos estão com o build quebrado.
* [**Selenium**](http://seleniumhq.org/): também desenvolvida pela ThoughWorks, provê maneira de realizar testes de integração em projetos Web, simulando a navegação do usuário pelas páginas do site, realizando diversas ações como clicar o mouse, prencher campos, etc... Usado para validar se o resultado de determinadas ações do usuario correspondem às que esperamos.
* [**Xvfb**](http://www.xfree86.org/4.0.1/Xvfb.1.html): Servidor X11 que permite realizar todas as operações gráficas em memória, não exibindo nada na tela. Com essa ferramenta, é possível rodas os testes com Selenium sem ter as janelas do Firefox pulando na tela do nosso servidor a cada teste. Além disso, podemos rodar os testes mesmo em ambientes sem X instalado (como é o caso de servidores, por ex).

Ainda usamos plugins Maven e Ant para a geração de relatórios de cobertura e testes, que publicamos a cada novo build. Assim, conseguimos além de garantir que o código está todo integrado, garantir que está com uma cobertura adequada; afinal, de que adianta ter o sistema sem testes falhando, se ele possui menos testes que o necessário?

![JUnit Report](assets/integracao-continua/junit-300x52.png "JUnit Report")

![Cobertura Report](assets/integracao-continua/cobertura-300x272.png "Cobertura Report")

Por fim, pela experiência que temos no uso dessas ferramentas no desenvolvimento de aplicações internas, open source e em nossos clientes, podemos afirmar que elas de fato nos trazem uma produtividade antes inalcançável. Se você pensa em começar a utilizar alguma ferramentas de integracão contínua, vale a pena dar uma olhada nessas que indicamos. No entanto, existem várias outras disponíveis, e é questão de escolher a que melhor se adapta às suas necessidades. Não deixe de comparilhar suas experiências..

# ntegração Contínua e o processo Agile

# Integração Continua - Builds rápidos com Grids e paralelismo

Como já comentamos em um [post anterior](https://blog.caelum.com.br/integracao-continua/), aqui na Caelum fazemos a [Integração Contínua](http://martinfowler.com/articles/continuousIntegration.html) das nossas aplicações, com a ajuda de algumas ferramentas, entre elas o [Selenium](http://seleniumhq.org/" \t "_blank), para os testes de integração das aplicações.

Um dos princípios de Integração Contínua é que, no final do processo de build, tenhamos um produto pronto pra ir pra produção. Para garantir que a aplicação está nesta situação, precisamos criar diversos tipos de testes automatizados, como os unitários e os de integração.

Testes de integração são aqueles que testam funcionalidades completas, ou seja, testam a integração de vários módulos do seu sistema para garantir que a funcionalidade desejada está completa. Em aplicações web, esse tipo de teste costuma envolver abertura de browsers e simulação da interação do usuário com o sistema.

O problema desses testes fazerem parte do build da aplicação, é que eles tendem a demorar demais à medida que a aplicação cresce, ferindo outro princípio da Integração Contínua: seu [build deve ser o mais rápido possível](http://martinfowler.com/articles/continuousIntegration.html" \l "KeepTheBuildFast" \t "_blank). Não adianta nada ter um servidor de Integração Contínua, se seu build demora demais, por exemplo mais de 30 minutos. Imagine que você só descobre que a última mudança que enviou para o controle de versão (commit) quebrou alguns pontos de seu programa somente meia hora depois de iniciar uma nova tarefa. Sua mente, como desenvolvedor, já está muito longe do que executou até meia hora atrás e focada em um problema totalmente distinto.

[Dez minutos](http://jamesshore.com/Agile-Book/ten_minute_build.html) é um bom limitante superior pro tempo do seu build, fazendo com que o feedback seja rápido, assim um build quebrado tende a ser corrigido imediatamente, e a aplicação a estar sempre no estado pronto pra deploy o tempo todo.

Mas como fazer um build de menos de dez minutos, quando só os meus testes de integração demoram mais de meia hora? Uma alternativa é tirá-los do processo de build, e rodá-lo só no fim do dia, ou a cada hora. O problema é que perdemos o feedback rápido dos testes de integração, caindo em uma situação ainda pior que a citada acima, com o feedback de 30 minutos.

Outra alternativa, bem mais interessante, é rodar seus testes de integração em paralelo. Aqui na Caelum usamos algumas ferramentas para conseguir rodar os testes dessa maneira. A primeira delas é o [Parallel Junit](https://parallel-junit.dev.java.net/" \t "_blank), capaz de rodar em paralelo um conjunto qualquer de testes compatíveis com JUnit. Para aqueles que usam o TestNG, existe um suporte natural à essa funcionalidade, que permite somente passar parâmetros na sua tag do ant ou maven.

De qualquer maneira, não adianta rodar somente uma instância do Selenium e tentar rodar os testes em paralelo. Então usamos o [Selenium Grid](http://selenium-grid.seleniumhq.org/" \t "_blank), que permite subir mais de uma instância do Selenium Server na mesma máquina, ou melhor ainda, em várias máquinas, deixando isso transparente para os testes que usam o Selenium. De uma maneira ainda mais emocionate, podemos levantar diversos browsers distintos em máquinas com sistemas operacionais diferentes, possibilitando rodar os testes em paralelo em ambientes como Windows e Linux.

No Selenium Grid temos duas partes importantes: O **Selenium Hub**, que vai responder aos comandos do selenium emitidos pela sua aplicação, e delegá-los para algum dos **Remote Controls**, que executarão os mesmos no selenium de verdade. Em outras palavras, o Hub funciona como um proxy para cada comando que o teste deseja executar no browser, delegando essa requisição para algum Remote Control ocioso.

Um grande problema do Selenium Grid é a sua usabilidade. Para configurá-lo, precisamos subir o Selenium Hub em uma máquina, depois ir em cada máquina que rodará os Remote Controls, e subir um por um na mão, usando uma task pronta do Ant. Chato? Mas ainda não é o maior problema...

Se a máquina que tem o Hub cair, precisamos ir em todas as máquinas que tem os Remotes, e registrar tudo novamente. Se uma máquina que tem os Remotes cair sem executar um shutdown limpo, não desregistrando seus remotes, precisamos reiniciar tudo de novo, pois se um teste tentar usar o Remote "fantasma", vai dar erro. Por fim, se um teste abrir um Selenium, e esquecer de fechar, o Remote Control associado a esse Selenium ficará travado para sempre, e então você terá que reiniciar todos os programas novamente.

Por causa disso, eu e o Guilherme Silveira resolvemos hackear o Selenium Grid e resolver esses problemas. Criamos então um [fork do projeto original no github](http://lucascs.github.com/selenium-grid" \t "_blank), e um novo projeto, o [Selenium Box Agent](http://lucascs.github.com/selenium-box-agent" \t "_blank).

A idéia desse novo projeto é a seguinte: Em cada máquina onde serão rodados os Remote Controls, deixamos uma instância do Box Agent rodando. Na máquina onde rodará o Hub, rodamos o projeto do Grid modificado. E então é só ir na página de console do Hub (geralmente em <http://localhost:4444/console>, substituindo localhost pelo ip do seu servidor) e registrar os Box Agents, pedindo para iniciar os Remotes na mesma página, sem necessidade de linha de comando, tasks do ant, nem qualquer outro trabalho manual e repetitivo. Apenas formulários e links na página do console.

![Página Principal do Hub modificado](assets/integracao-continua-builds-rapidos-com-grids-e-paralelismo/hub1-300x187.png)

``` ```caption id="attachment\\_569" align="alignnone" width="300" caption="Box adicionado"

![box adicionado](assets/integracao-continua-builds-rapidos-com-grids-e-paralelismo/hub2-300x187.png)

``` ```caption id="attachment\\_570" align="alignnone" width="300" caption="Remote Control Adicionado"

![Remote Control Adicionado](assets/integracao-continua-builds-rapidos-com-grids-e-paralelismo/hub3-300x187.png)

Se a máquina do Hub cair, quando ela voltar levantará automaticamente os Remotes de todos os Boxes registrados. Se uma máquina de Box cair, o Hub desregistrará automáticamente todos os seus Remote Controls, e quando ela voltar o Hub registra todos eles automaticamente também. Se um teste esquecer de devolver um Selenium, você pode desregistrar o RemoteControl associado, e pedir pra registrá-lo novamente, direto da página de console.

Um dos nossos builds, que demorava cerca de 50 minutos quando os testes eram rodados serialmente, demora por volta de 8 minutos quando rodado em paralelo, usando 7 threads simultâneas de testes. Se a aplicação crescer mais, é só registrar mais Remote Controls e aumentar o número de Testes simultâneos. E pronto, conseguimos o build de dez minutos, sem sacrificar os testes de integração.

O resultado? Uma equipe mais propensa a corrigir seus próprios erros a medida que eles são cometidos, sem ter que esperar o feedback de um cliente, possivelmente furioso, sobre algo que foi quebrado na última entrega.

Aqui na Caelum tentamos sempre encontrar alguns projetos com os quais podemos contribuir, seja com comentários, documentação ou código e ter sido capazes de criar uma extensão para o Selenium Hub é mais uma maneira que encontramos para compartilhar algo criado por nós com aqueles que procuram e precisam de uma solução similar.

# Integração contínua: deploys e aprovações sem dor de cabeça para o cliente

Em 2008 comentamos sobre [a importância de integração contínua](https://blog.caelum.com.br/integracao-continua/) no processo de receber feedback rápido sobre suas mudanças em um sistema e depois sobre os [problemas que surgem](https://blog.caelum.com.br/integracao-continua-builds-rapidos-com-grids-e-paralelismo/) quando um sistema possui baterias de teste muito grandes e complexas.

Um das grandes vantagens da agilidade consiste em poder efetuar mudanças sem medo e receber as respostas rapidamente em relação aos bugs que introduzimos no sistema, [evitando mais erros](http://www.jacoozi.com/blog/?p=11) e, para atingir esse objetivo, um servidor de [integração contínua](http://martinfowler.com/articles/continuousIntegration.html) deve integrar o nosso código com aquele existente e rodar a bateria de testes automatizados a cada commit, criando um relatório do que foi feito que pode ter quebrado a aplicação.

Existem diversos níveis de adoção de um servidor de integração contínua em um projeto, indo do mais básico que envolve a simples [compilação de um projeto](https://www.alura.com.br/artigos/o-que-e-compilacao) (quando a linguagem é compilada) e a execução de testes unitários, até cenários mais complexos. Uma das principais dores de cabeça que um cliente enfrenta, mesmo em empresas que adotam [metodologias ágeis](http://agilemanifesto.org/) como [XP](http://www.extremeprogramming.org/) e [Scrum](http://www.caelum.com.br/curso/pm-83-gerenciamento-agil-projetos-scrum/), está ligada ao produto ser colocado em produção e não funcionar: a famosa frase "mas no meu computador funcionou".

Um cenário clássico envolve o desenvolvimento de testes unitários, que são rodados na máquina do desenvolvedor e no servidor de integração contínua. O desenvolvedor chama então o cliente para testar em sua máquina, que aprova a funcionalidade e depois de duas semanas é feito o deploy da aplicação quando, para a surpresa do cliente, ela não funciona como esperado.

Acontece que a máquina de desenvolvimento em geral não possui a mesma configuração - software de banco de dados, servidor, proxies, firewall ou até mesmo os dados contidos no banco - que o sistema em produção, e mudanças do gênero podem significar resultados completamente diferentes para o cliente que, no fim da iteração, não obtém o valor que esperava.

A solução está em criar um sistema de homologação e diversas empresas adotam essa prática. A dificuldade encontrada nessa etapa é a da não automatização do processo de réplica da estrutura e dados de produção para homologação. Dados antigos ou inadequados no ambiente de aprovação facilitam um aceite errôneo por parte do cliente no momento do teste. Na Caelum e em nossos clientes que possuem sistemas que efetuam deploy diversas vezes em um ano, para garantir a competibilidade da empresa, é comum automatizar o processo de deploy ao ponto de [maximizar as semelhanças entre produção e homologação](http://martinfowler.com/articles/continuousIntegration.html#TestInACloneOfTheProductionEnvironment), e minimizar as idas e vindas de uma funcionalidade.

Passo a passo junto com o cliente, cada fase é incluída no [pipeline de integração contínua](http://www.thoughtworks-studios.com/cruise-release-management). Primeiro o projeto é compilado, depois os testes unitários são executados. Essas duas fases do pipeline são padrão para linguagens compiladas como Java. Em algumas empresas devido ao código legado ter um acoplamento muito alto, essa primeira fase já envolve mudanças na maneira de trabalhar e na qualidade do código gerado pelos desenvolvedores.

As fases seguintes variam bastante de acordo com projeto, mas para melhorar o deploy que minimize o número de tentativas de aprovação é importante adicionar três fases: testes [end-to-end automatizados](http://www.slideshare.net/spriebsch/end-to-end-web-testing-with-selenium-presentation" \t "_blank), deploy para homologação e para produção em um clique.

Na primeira fase, os testes executam as funcionalidades como o cliente faria e cobrem a maior quantidade de funções plausíveis de automatização. Na segunda fase, os desenvolvedores podem optar por deployar para um sistema de homologação, onde o cliente aprovará com os dados copiados de produção (alterações ligadas a sigilo de dados devem ser aplicadas).

Por fim, quando o sistema está aprovado em homologação, com um clique o desenvolvedor efetua o deploy para produção. Nesse instante, os sistemas são restartados, de preferência com [o uso de load balancers](https://blog.caelum.com.br/melhorando-o-guj-jetty-nio-e-load-balancing/) que permitem o restart sem a queda da aplicação, e o cliente está pronto para usar as funcionalidades novas.

Toda essa automatização tem um custo inicial de desenvolvimento que é compensado com a minimização de erros humanos no processo de deploy e na lentidão do mesmo. Muitas vezes o processo de deploy para qualquer um dos dois ambientes não é feito tão frequentemente devido ao processo ser demasiadamente complexo para ser executado manualmente. A chance de funcionar em produção é maximizada uma vez que a mesma foi testada em homologação com uma cópia mais fidedigna do sistema de produção possível: menos reclamações e tempo perdido pelo cliente.

Note que nenhuma ferramenta específica é necessária para adotar essa prática, basta mudar o método de trabalho.

<https://martinfowler.com/bliki/ContinuousDelivery.html>