Diretrizes Desenvolvimento

Orientações básicas para execução de projetos

Se existirem razões práticas que obriguem o desvio de algumas dessas diretrizes durante o desenvolvimento de um projeto, nesses casos é possível revisar esse documento ou deixar explícito na documentação de tal projeto as razões para o desvio.

### WIP Warning

As informações deste documento são “vivas” e podem ser modificadas em colaboração com todas as equipes.

# Sumário

[WIP Warning](#_ayxq6cqghg9x)

[Sumário](#_1aqpnqo1ec5j)

[Processos e práticas](#_fkbmys7blkef)

[Arquitetura](#_lhu6d2bh25en)

[APIs](#_6z8h019bmxfb)

[Logging](#_x5zxg1l8mgsr)

[Configurações](#_i12luqtp8zsl)

[Implementação](#_nqkvv6jh0mng)

[Segurança](#_yum3jf9o3ovo)

[Monitoramento](#_il4cti8smz8c)

[Banco de Dados](#_c8irp9lv38vf)

[Interface](#_o1e6tgmt4udk)

[Emails](#_n6g3k6ik4shr)

[Documentação](#_c6lxturw8ipe)

# [Processos e práticas](#_fkbmys7blkef)

Diretrizes relacionadas ao processo de desenvolvimento

* Práticas e métodos:
  + Code Review - deve ser incentivada mas não obrigatória.
    - Desenvolvedor convida outros desenvolvedores relevantes para revisar o seu próprio código.
    - Desenvolvedor relevante é qualquer um que tenha um bom conhecimento do sistema desenvolvido.
    - Pode-se usar o sistema de checklist do Github nos Pull Requests para convidar outros desenvolvedores.
  + Pair Programming deve ser incentivado. E é importante incentivar o Pair Rotation.
  + Meta-Documentação sobre práticas e métodos (aka RTFM) - documentação obrigatória abordando todos os assuntos necessários para subsidiar o trabalho de um membro novo na equipe.
  + Usar [Github Flow](https://guides.github.com/introduction/flow/index.html) - recomendado para equipes em estágio inicial.
  + Deve se usar o inglês no código fonte das aplicações (identificadores, docstrings, etc). Strings apresentadas para usuários devem aparecer em português (hardcoded ou com i18n).
  + Escrever [boas mensagens de commit](https://github.com/erlang/otp/wiki/Writing-good-commit-messages) e padronizá-las em um único idioma por projeto (preferencialmente em inglês se toda a equipe se sentir confortável com esse idioma).
    - <http://chris.beams.io/posts/git-commit/>
  + Infrastructure Automation:
    - Continuous Integration - recomendado (ver tópico sobre [Testes e Integração Contínua](#_wx0vnc26wsdm)).
    - Continuous Delivery (ex. one-step deployment) - recomendado.
    - Facilidade para configurar um ambiente local de desenvolvimento - recomendado reaproveitamento do sistema de Continuous Delivery.
  + Testes Automatizados - necessário.
* Visão de Negócio
  + Todos os desenvolvedores devem sempre considerar a visão macro dos negócios da empresa ao tomar uma decisão.
* Vocabulário comum
  + Glossário

# [Arquitetura](#_lhu6d2bh25en)

* KISS (pequenos componentes altamente especializados).
* Serviços disponíveis via APIs (ver seção APIs).
* Reforço de Regras de negócio no RDBMS (ex. constraints).
* Dependências Mínimas:
  + Diminuir número de dependências de um projeto.
  + Pensar muito sobre a necessidade de adicionar mais um elemento na pilha de software da solução.
  + Menor número de “partes móveis”.
  + Adotar novas ferramentas apenas quando forem benéficas para o projeto (e não porque há um desejo do desenvolvedor/equipe para usar essa ferramenta).
  + Aproveitamento de serviços já disponibilizados pelos fornecedores principais da Titans (ex. AWS) e/ou pela pilha de software já adotada pela empresa.
* Serviços de terceiros
  + Sempre considerar e especificar tratamento para cenários de indisponibilidade desses serviços nas integrações.
* Integrações
  + Analisar a viabilidade/necessidade de uso do Circuit Breaker pattern (ferramentas de [orquestração de integrações](https://github.com/Netflix/Hystrix)). Segue um guideline com critérios para a adoção ou não deste pattern: <http://herzberg.ca.sandia.gov/guidelines.shtml>
  + Qualquer integração interna ou externa deve ter tempo de *timeout* definido.
  + Qualquer integração interna ou externa deve ter política de retentativa definida. A política de retentativas deve ser baseada em SLA dos sistemas envolvidos e criticidade da operação.
  + Integrações devem ser monitoradas quanto à seu estado de funcionamento (ver tópico monitoramento/health check) e quando à sua operação (ver tópico logging).
  + Fazer tratamento adequado de falhas em integrações que fazem parte de uma transação (rollback/revert).
  + Definir política de manutenção da integração. SLA, como tratar manutenção.
  + É “critério de pronto“ para qualquer integração que ela esteja com todos os pontos acima implementados e documentados.
* Manutenção
  + Quando um sistema estiver em manutenção deve avisar seus clients apropriadamente (ex. HTTP code 503 com informações de hora de fim no cabeçalho).
  + Um cliente de um sistema em manutenção deve ser capaz de se recuperar e tomar as ações necessárias: Contenção de retentativas até o restabelecimento do serviço; aviso apropriado ao usuário. O ideal é que isto ocorra sem que seja necessário um “deploy de manutenção”.
  + Os clients devem definir política de retentativa levando em consideração a existência do processo de manutenção.
* Auditabilidade
  + Todas as transações devem ser auditáveis. Devem conter dados como: Timestamp, Operação, Origem, Meio, Campanha, Credenciais, etc. Utilizar um sistema de [rastreamento de transação](https://devcenter.heroku.com/articles/http-request-id) e gerar registros de transação com esse identificador é obrigatório .
  + A auditabilidade deve ocorrer inclusive entre as integrações. Assim se um request iniciar um processo de compra, sistemas como o TCM devem também receber este identificador da transação.
  + As informações armazenadas devem tornar possível a replicação da transação a qualquer momento obtendo-se o mesmo resultado.
* Backoffice
  + Nehuma ferramenta de backoffice deve estar exposta no mesmo domínio da aplicação (API, WEB, etc).
  + Deve ter uma política de restrição de acesso por usuário e senha e apenas a partir da rede da Titans.
* Identificação de usuário
  + O identificador do usuário deve ser único e imutável para toda a empresa, ou seja, não pode ser o MSISDN.

# [APIs](#_6z8h019bmxfb)

Diretrizes para uniformização de APIs

* Devem ter changelog.
* APIs devem ser versionadas:
  + APIs devem ter apenas versões “major” (v1, v2, …).
  + Mudar a versão sempre que houver quebra de compatibilidade retroativa.
  + Manter a versão anterior dentro de um plano de “deprecation” definido previamente em cada projeto.
  + APIs HTTP devem usar prefixo no Path da URL (ex. /v1/, /v2/, …)
  + O modelo de versionamento deve ser adotado consistentemente em todas as APIs de um mesmo projeto.
* Requests e Responses de APIs não devem ser internacionalizadas. Devem ser mantidas sempre em inglês.
* APIs HTTP:
  + Secure HTTP (https://…) obrigatório
  + Preferência por APIs RESTful
    - Suporte a todos os verbos definidos pelo HTTP
    - Suporte ao header X-HTTP-Method-Override.
  + Documentação (ver tópico sobre documentação)
  + Preferencialmente Síncronas para resposta < 200ms. Otimizar antes de tornar a API assíncrona.
  + Assíncronas (tempo de respostas >200ms ). Respondem 202 Accepted ao invés de 200 OK.
  + Recomenda-se que APIs assíncronas executem uma requisição (webhook) em uma URL informada pelo cliente para avisá-lo sobre a conclusão da operação.
  + Analisar a viabilidade de criar APIs “discoverable” (HATEOAS).
  + Serialização:
    - JSON obrigatório.
    - Serialização binária recomendada para APIs internas (Protocol Buffer, Thrift, etc).
    - Outros formatos opcionais (ex. XML, etc.).discoverable
  + Authenticação para Backoffice APIs:
    - <https://www.owasp.org/index.php/Authentication_Cheat_Sheet>
    - Basic HTTP Auth
    - Signed Token (assinatura do request)
    - Web Token (HTTP Header: Authorization: Token …)
    - [OAuth2 Client Credential Flow](https://tools.ietf.org/html/rfc6749#section-1.3.4)
* APIs baseadas em Filas (operações sem garantia ou com tempo de resposta muito grande)
  + Filas Assíncronas (ex. Celery/RabbitMQ/AWS SQS)
* Bibliotecas Clientes para APIs
  + Todas as libs devem ser versionadas segundo as diretrizes de versionamento semântico <http://semver.org/> ignorando apenas os sufixos como: pre, rc, alpha… Sendo assim, usaremos apenas minor.minor.patch. Todas as alterações de libs devem ser acompanhadas pela atualização da versão. Mais detalhes no [documento de versionamento semântico](https://docs.google.com/a/titansgroup.com.br/document/d/1LJWbmZgcd1mKmmxo5MPB8YaDhY9TBt6XBgISDEj-H5I/edit?usp=sharing).
  + Toda lib deve ter um changelog atualizado. Podemos usar como referência as definições do projeto GNU <https://www.gnu.org/prep/standards/html_node/Change-Logs.html>
  + Bibliotecas devem gerar logs de comunicação.
    - Desativado por padrão (ver [exemplo](https://github.com/titansgroup/tnt-sync-lib/blob/master/tnt_sync_lib/__init__.py))
    - Ver tópico de Logging deste documento
  + Toda comunicação precisa definir um Timeout levando em consideração o SLA e as integrações das quais este serviço depende.
    - ex: Serviço B acessa sincronamente serviço C para atender uma requisição do serviço A. O Timeout de A deve ser superior ao timeout de B ao acessar C.
  + Preferencialmente as bibliotecas não devem tratar retry. Isso pode ser tratado pela aplicação cliente da biblioteca.
  + Tratar exceção em vários níveis:
    - APIError = Qualquer tipo de erro
    - CommunicationError(APIError) = Erros de Comunicação com API. Timeout, SSLError, Unknown Host, ...
    - ServerError(APIError) = Erros 5XX ou de parsing da resposta
    - ClientError(APIError) = Erros de acesso à API (4XX)
    - *Model*Error(ClientError) = Erros específicos da API (ex. UserNotFound)
  + Sugestão de API mínima para bibliotecas cliente de webservices (pode-se criar camadas extras de abstração às libs mas seria bom ter essa API mínima):

client = APIClient(credentials=...) # never throw exception

client.connect() # optional method for queue APIs

try:

response = client.method(arguments)

except ...APIErrors...:

...

print(type(response)) # unserialized JSON object as list/dict

* Não devem expor detalhes de implementação (ex. id/pk de um registro no banco de dados na URL).
* APIs Públicas devem ser protegidas contra uso abusivo.
* Recomenda-se a proteção contra uso abusivo também nas APIs Privadas e de parceiros.

# [Logging](#_x5zxg1l8mgsr)

Diretrizes para produção de logs

* Práticas:
  + Usar o módulo [logging](https://docs.python.org/2/library/logging.html) do Python (ou [logbook](https://github.com/mitsuhiko/logbook))
  + Não usar “print”.
  + Usar sempre UTC como Timestamp.
  + Usar sempre console (stdout ou stderr) como handler para logs.
  + Logs de uma transação devem ser identificados para torná-los rastreáveis (traceability). Exemplo: [flask\_trace](https://github.com/canassa/flask-trace).
  + Seguir recomendações em:
    - <http://12factor.net/logs>
    - <http://dev.splunk.com/view/logging-best-practices/SP-CAAADP6>
  + Não fazer logs multilinha (ex. tracebacks deveriam ser enviados para o Sentry com um ID de rastreabilidade e nunca para um log).
  + Nenhuma informação sensível deve ser logada. Nunca. (ex. senhas ou chaves de acesso).
  + Logs não devem ser enviados para sistemas de registro de erros como o Sentry. Logs devem ser enviados para sistemas de gerenciamento de logs.
  + Deve-se dar preferência para a representação dos objetos nas strings mensagens (ex. repr(obj) ou "%r" % (obj,) ou ainda "{!r}".format(obj)) para evitar problemas de codificação de caracteres.
* Formato:
  + LEVEL - nível de log
  + TIMESTAMP - asctime
  + UUID - UUID da transação (ex. request/response é uma transação)
  + MODULE - módulo do código onde o log foi gerado (ex. logger = logging.getLogger(\_\_name\_\_))
  + MESSAGE - formato key1=value1, key2=value2, keyN=valueN, …
* Níveis
  + **DEBUG**: informações detalhadas e avulsas de uma transação e suas sub-transações incluíndo valores de variáveis internas locais de um método, por exemplo. Deve ser ativada nos ambientes Dev e Local ou durante sessões de debugging em ambientes de produção. Ex: (method=login, username=foo, apiurl=https://…, response=...)
  + **INFO**: informações compiladas e agregadas de uma macro-transação. Ex: (username=foo, operation=login, response=success, retry=1 time). Esse nível de log está habilitado em Produção, UAT, Dev e Local. Se o volume de logs gerados em produção se transformar em um problema pode-se reavaliar a geração deles neste ambiente.
  + **WARNING**: Se ocorreu algo de errado ou fora do comum durante a transação mas você consegue se recuperar dessa situação use este nível de log. Ex. (method=login, username=foo, response=success, **retry=3 times**). Esse nível de log deve estar habilitado em todos os ambientes.
  + **ERROR**: Se ocorreu um erro durante a transação mas você sabe lidar com esse erro respondendo ao cliente que esse erro ocorreu. Deve ser considerado apenas os erros dos sistemas e subsistemas do backend e não erros causados pelos clientes (ex. usuário/senha inválidos não devem ser logados) Ex. (username=foo, operation=login, response=fail, errorcode=connection\_error, retry=5 times). Esse nível de log deve estar habilitado em todos os ambientes.
  + **CRITICAL**: Esse tipo de erro pode ser logado mas ele também deve ser lançado como exceção do sistema. Ele acontece em casos de erros irrecuperáveis ou não tratados (ex. Bugs do sistema). Esse tipo de erro é o que retorna um código 5XX em uma transação HTTP e envia o traceback para um sistema de registro de erros como o Sentry.

# [Configurações](#_i12luqtp8zsl)

Como lidar com configurações da aplicação

* Evitar arquivos de configuração separados para cada ambiente
* Configurações via variáveis de ambiente (<http://12factor.net/config>).
* Usar bibliotecas de desacoplamento de configuração (ex. [prettyconf](https://pypi.python.org/pypi/prettyconf))
* Colocar nas configurações do sistema apenas configurações “de software”. Configurações relacionadas a regras de negócio devem, preferencialmente, ser consideradas como dados do sistema e configuráveis, por exemplo, via interface administrativa.
  + Configurações que exigem um número muito grande de deployments causados por mudanças de configurações nas regra de negócio são candidatos fortes à deixar os arquivos de configuração do projeto e serem acessíveis via configurações.

# [Implementação](#_nqkvv6jh0mng)

Desenvolvimento de código e implementação

* Usar semantic versioning (<http://semver.org/>)
* Identificadores devem ter nomes em inglês.
* Utilizar sempre as versões estáveis mais atuais de linguagens, frameworks e bibliotecas priorizando essas atualizações sempre que surgirem.
* Timestamps devem ser armazenados em UTC
* Mensagens e Strings devem ser escritas em inglês e marcadas para tradução (exceto para APIs).
* Seguir (e garantir) o uso da PEP8 (com uma única exceção do tamanho da linha)

# [Segurança](#_yum3jf9o3ovo)

Diretrizes básicas de segurança

* Acompanhar e aplicar atualizações de segurança em nosso software e em suas dependências.
* Limitar acesso de APIs públicas para evitar abuso. (Ver item de APIs públicas)
* Utilizar apenas sistemas e métodos de segurança conhecidos.
* Não amarrar as medidas de segurança à infra-estrutura (ex. bloqueios por IP).
* Não usar chaves da AWS nos projetos nem nas configurações
* Atenção para vetores de ataques de segurança
  + HTTP Error Handler - Utilizar página estáticas no lugar do tratamento padrão dos frameworks pois, caso exista um erro na própria página/template de erro, isso poderá ser explorado para causar ataques de DoS. Para internacionalizar essas páginas pode-se adotar alguma [configuração específica do servidor web](http://devblog.springest.com/using-localized-500-error-pages-with-nginx/).

# [Monitoramento](#_il4cti8smz8c)

* Operacional
  + Pingdom
  + Sensu
  + Health check
    - Todos os serviços devem responder pelo endpoint “/health” (sem trailling backslash).
    - O acesso a este endpoint deve ser restringido por algum tipo de autenticação (ex. Basic HTTP Auth). Não é necessário restringir acesso por IP de origem;
    - Deve retornar um JSON contendo o status do serviço em questão e de todos os subserviços. Ex: {“status”: “OK”, “systems”: {“database”: “OK”, “mitelcel”: “OK”, “tcm”: “ERROR”}}
    - Não deve conter informações sensíveis (ex. Hostnames, usuários, database, …)
  + New Relic
    - Apdex
    - Overall Performance
    - Error
* Desenvolvimento
  + New Relic
    - Specific Performance Aspects (ex. slow queries, profile, etc…)
    - Errors and Exceptions
  + Sentry
    - Erros de desenvolvimento (exceptions/tracebacks).
    - Logs de erro não devem ser enviados para o Sentry (ver tópico sobre Logging).

# [Banco de Dados](#_c8irp9lv38vf)

* Tabelas
  + Toda tabela deve ter chave primária
  + A chave primária deve se chamar id
  + O nome da tabela deve ser escrito em underscore case (ex. titans\_products)
* Campos/Colunas
  + O nome do campo deve ser escrito em underscore case
  + O nome de uma foreign key para a chave primária de outra tabela deve ser <nome\_da\_outra\_tabela>\_id.
  + Campos varchar devem ter o tamanho definido coerente com o tamanho máximo dos dados que irá conter. Campos varchar de tamanho inadequado podem causar um consumo de memória extra no servidor e desabilitar certas otimizações em consultas (ex. tabelas temporárias em memória). Alguns SGBD desabilitam otimizações no uso/criação de índices dessas colunas.
  + Se um campo ou um conjunto de campos tem restrição de valores (unicidade para múltiplos campos - candidate key, range, tamanho, etc) e o banco suporta a validação, implemente-a.
  + Campos varchar não devem ultrapassar 255 caracteres, pois o MySQL só indexa 255 caracteres. Pela mesma razão não é possível configurar unicidade para campo destes.
* Indexação
  + Todo campo utilizado para busca sistêmica (implementada em algum método ou alguma busca implicita gerada pelo ORM) deve ser indexado.
  + Não indexar campos sem necessidade.
  + Avaliar os vários tipos de índices oferecidos pelo SGBD para decidir qual é o mais adequado para a aplicação.
* Práticas e Cuidados
  + Muito cuidado com comandos ALTER TABLE em tabelas grandes. Ele pode demorar e, certas operações em certos SGBDs, pode bloquear (lock) o acesso à tabela.
  + Criação de índices em particular tem um custo bem alto e pode se valer desta técnica. Embora o suporte de MySQL tenha melhorado alguns comandos ainda fazem lock na tabela.
  + Uma técnica útil é sincronizar a tabela original com uma nova tabela com a estrutura nova e depois dropar a original e renomear a nova. Cuidado com Foreign Keys. Faça Backup antes, pois alterações DDL no MySQL não são transacionais.
  + É preciso parar o sistema por alguns instantes para as operações de dropar e renomear.

Técnicas para lidar com migrações de dados ou mudanças estruturais significativas em produção (em ordem de preferência):

1. No caso de migração de sistemas pode sincronizar as tabelas dos dois sistemas antes com um processo de ETL e, sincronizadas as tabelas, virar a chave.
2. Quando o cenário anterior não é viável:
   1. Parar o sistema
   2. Fazer o backup
   3. Fazer as alterações/migrações necessárias
   4. Iniciar o sistema

# [Interface](#_o1e6tgmt4udk)

*Diretrizes relacionadas ao processo de desenvolvimento de Interface*

* Padrões de Desenvolvimento de Interface (Wiki - <http://bit.ly/1wHhTr7>)
* Ambiente de Interface Básico nos Projetos (Wiki - <http://bit.ly/1wHhTHN>)

# [Emails](#_n6g3k6ik4shr)

Toda a comunicação por e-mail com assinantes deve ter tratamento de bounces. Isto permite quailificar as bases que temos e evitar interrupções e problemas com o domínio utilizado em decorrência da entrada em blacklists.

Todo o e-mail recebido deve ser validado antes de ser utilizado. Se a regra de negócio não permitir a validação obrigatória o e-mail não poderá ser utilizado para envio de mensagens.

# [Documentação](#_c6lxturw8ipe)

Todas equipes e projetos precisam dedicar atenção especial à produção de documentos usando ferramentas e formatos descritos abaixo.

* Tipos de Documentos
  + Documentação sobre Equipes e processos
    - Informações necessárias para o funcionamento das equipes
    - Dados gerais sobre projetos em que cada equipe trabalha
    - Dados sobre os integrantes das equipes (ex. dados de contato)
    - Lista de serviços (ex. slack, mumble, github, etc) que os membros da equipe precisam ter acesso
    - Listas de discussão que o integrante da equipe precisa participar
    - Horários de cerimônias e reuniões (ou link para uma agenda)
  + Documentação de negócio dos Projetos, Sistemas e Serviços
    - Descrição do projeto/sistema.
    - Contexto do projeto nos negócios da Titans.
  + Documentação do Projeto, Sistema ou Serviço
    - Documentação detalhada sobre o funcionamento do sistema, os seus módulos e componentes e como eles interagem entre si.
    - Referência para uso do Projeto, Sistema ou Serviço por outras equipes ou terceiros.
* Ferramentas de Documentação
  + Wiki (<http://wiki.corp.titansgroup.net/index.php/>) - ideal para documentar informações sobre Equipes e processos e também para documentar (ou referenciar) informações sobre contexto de negócio de cada Projeto, Sistema ou Serviço.
  + Sphinx (<http://sphinx-doc.org/>) - cada projeto deverá disponibilizar um diretório (./docs/) contendo a documentação completa sobre o projeto usando uma instalação do Sphinx. Deve-se usar o tema ??? em todos os projetos. A documentação completa deve ter items como:
    - Documentação discursiva (obrigatório)
    - Tutorial de uso (recomendado)
    - Referência de APIs para serviços, bibliotecas ou frameworks (obrigatório)
    - Referência de APIs internas para outros tipos de projeto (recomendado)
  + O sistemas de CI da Titans deve sempre gerar a documentação e exportar o resultado em um endereço conhecido e referenciado na Wiki do projeto.
* Dicas para produção de documentos
  + Usar ferramentas colaborativas online para prototipar e esboçar documentos (ex. Google Docs, Apiary, …) e posteriormente migrar o conteúdo para os locais indicados pelas diretrizes.
  + Usar extensões de autodoc para o Sphinx gerar a documentação de referência a partir das docstrings no código.

# Testes e Integração Contínua

* Todos os projetos da Titans devem ser desenvolvidos acompanhado de testes automatizados.
* Nenhuma técnica (ex. TDD) ou ferramenta (ex. pytest, nose, …) é obrigatória. Cada time tem autonomia para escolher seu conjunto de ferramentas.
* Os ambientes de testes devem emular o ambiente de produção:
  + Usar o mesmo SGBD (MySQL, PostgreSQL, …)
  + Usar o mesmo serviço de cache (memcached, Redis, …)
  + Usar o mesmo broker (RabbitMQ, SQS, …)
  + ...
* Desenvolvedores podem ter esses serviços instalados diretamente em suas workstations ou usar sistemas de virtualização/containers como Vagrant ou Docker.
* Desenvolvedores, localmente, podem usar configurações diferentes dos ambientes de produção para acelerar a execução dos testes (ex. usar sqlite em memória como banco de dados). Todavia, antes de submeter um Pull Request ou fazer Merge do seu branch com outro deve executar localmente os testes usando emulando um ambiente de produção.
* Todos os projetos devem ter e usar um sistema de CI. Atualmente a Titans oferece uma instância de [Jenkins](http://ci.titansgroup.net/) para uso de toda a empresa.
  + Os projetos novos utilizam Docker para execução dos testes. Essa prática é recomendável para criar ambiente mais parecidos com os que serão encontrados em produção.
* O CI permite integração com o Github permitindo que ele rode testes dos branches principais e dos Pull Requests submetidos para revisão.
* Existem muitas formas diferentes de se classificar e separar testes em software. Cada equipe pode usar a organização e classificação que preferir. Mas para simplificar este documento usarei uma divisão em dois tipos de testes: teste de código e teste de comportamento.
* O teste de código é aquele teste que é usado como ferramenta de apoio para o desenvolvedor. Geralmente são chamados de teste unitário, teste de integração, teste funcional, etc. Geralmente são usadas ferramentas do tipo xUnit para criação e execução desses testes.
  + Teste de models (ex. User.create(name="foo") -> assert user.name == "foo")
  + Testes de Web API (ex. self.client.post("/v1/user", data={"name": "foo"} -> assert self.client.get("/v1/user/1")["name"] == "foo")
  + Testes de interação do código (ex. change\_user.delay() -> assert user.name == "changed")
  + Esses testes devem ficar no mesmo repositório que o código testado. Mas a forma como organizar os testes é uma decisão da equipe.
  + Esses testes são coletados e executados, por padrão, pela ferramenta de testes do projeto.
  + Esses testes são executados no CI a cada commit/merge.
* O teste de comportamento é aquele que garante que a entrega atende os requisitos do cliente. Algumas pessoas chamam esses testes de “Behaviour Tests”. Alguns testes são escritos em linguagens específicas como [Gherkin](https://github.com/cucumber/cucumber/wiki/Gherkin) e outros são escritos em linguagens de programação usando bibliotecas como requests ou Selenium.
  + Testes de Web API (ex. requests.post("http://dev-machine/v1/user", {"name": "foo"}) -> assert requests.get("http://dev-machine/v1/user/1")["name"] == "foo")
  + Testes de App Web. Exemplo em Gherkin: Given I am on the "Home" page / When I click "Add Details" / Then I am on "Add Details" page
  + Testes de integrações com APIs de terceiros (ex. falar direto com endpoints de APIs de parceiros, testar cenários de sucesso e falha nessa integração, usar ferramentas de indução de falhas como [Vaurien](https://github.com/mozilla-services/vaurien)).
  + Esses testes devem ficar em um diretório exclusivo (ex. /behaviour/) para eles no repositório do projeto ou em um repositório exclusivo para eles.
  + Os testes devem interagir com um servidor rodando a aplicação (ex. LiveServerTestCase do Django e/ou usando Vagrant/Docker).
  + Esses testes não são coletados e executados por padrão pela ferramenta de testes do projeto. Para executá-los precisa-se solicitar explicitamente.
  + Esses testes não são executados no CI via hook de commit/merge. Deve-se executa-los com periodicidade regular (ex. 1x ao dia) no ambiente de CI.
  + Programadores e QA podem executar esses testes localmente quando desejar.
  + Programadores devem executar esses testes sempre que forem fazer merge de alguma modificação importante.
* Todos os integrantes da equipe são responsáveis pela qualidade dos testes de ambos os tipos.
* Se a equipe tiver profissionais de QA eles serão responsáveis pelos testes de comportamento e por cobrar os desenvolvedores pela criação de software testável.
* Em equipes com profissionais de QA os desenvolvedores não tem a responsabilidade de criar testes de comportamento mas devem colaborar com o trabalho desses profissionais.