

Apresentação do componente.**DESENVOLVIMENTO PARA SERVIDORES II (4) ³⁰**

Objetivos gerais. Desenvolver um site completo de *e-commerce* ou outro tipo de negócio na Internet usando uma linguagem apropriada a servidores, banco de dados e padrões de projeto.

Objetivos específicos. Implementar softwares do lado servidor e com uso de uma linguagem de programação e de padrões de projetos mais usuais como MVC, DAO, Composite, Singleton, entre outros.

Ementa. Conceitos e evolução das tecnologias de programação de servidores. Recursos da linguagem escolhida para servidores na Internet. Padrões de projetos. Integração com sistemas (Google Maps API, Twitter, entre outros)

Bibliografia básica

DEITEL, H; DEITEL, P. *Java – Como Programar*. São Paulo: Prentice-Hall do Brasil, 2010.

GAMMA, E et al. *Padrões de projeto*. Porto Alegre: Bookman, 2005.

MELO, A. A; LUCKOW, D. H. *Programação Java para a web*. São Paulo: Novatec, 2011.

Bibliografia complementar

DAIGNEAU, R. *Service design patterns*. Harlow (UK): Addison Wesley, 2011.

FREEMAN, E; FREEMAN, E. *Use a Cabeça! Padrões de Projetos*. 2, ed. Rio de Janeiro: Starlin Alta Consult, 2007.

HARTL, M. *Ruby on rails 3 tutorial*. Harlow (UK): Addison Wesley, 2011.

IERUSALIMSKY, R; CELES, W; FIGUEIREDO, L. H. *LUA programming gems*. Rio de Janeiro: LUA. Org, 2008.

RUBY, S; THOMAS, D; HANSSON, D. *Agile web development with rails*. New York: Oreilly & Assoc. 2010.

1-) Uso de celulares:

Para o bom andamento das aulas, recomendo que utilizem os celulares em *vibracall*, para não atrapalhar o andamento da aula.

2-) Material das aulas:

A disciplina trabalha com NOTAS DE AULA que são disponibilizadas ao final de cada aula, iremos utilizar o GitHub como repositório, o de nossa aula será:
<https://github.com/marcossousa33dev/FATECSRDSII202502>

3-) Prazos de trabalhos e atividades:

Toda atividade solicitada terá uma data limite de entrega, de forma alguma tal data será postergada, ou seja, se não for entregue até a data limite a mesma receberá nota 0, isso tanto para atividades entregues de forma impressa ou enviadas ao e-mail da disciplina. Qualquer problema que tenham, me procurem com antecedência para verificarmos o que pode ser realizado.

4-) Qualidade do material de atividades:

- Impressas ou manuscritas:

Muita atenção na qualidade do que será entregue, atividades sem grampear, faltando nome e número de componentes, rasgadas, amassadas, com rebarba de folha de caderno e etc. serão desconsiderados por mim.

- Digitais:

Ao enviarem atividades para o e-mail da disciplina, SEMPRE no assunto deverá ter o nome da atividade que está sendo enviada, e no corpo do e-mail deverá ter o(s) nome(s) do(s) integrante(s) da atividade, sem estar desta forma a atividade será DESCONSIDERADA.

5-) Critérios de avaliação:

Cada trimestres teremos as seguintes formas de avaliação:

- Práticas em Laboratório;
- Assiduidade;
- Outras que se fizerem necessário.

1. Introdução

Nessa fase do curso iremos abordar a ideia da programação no lado do servidor, o que podemos chamar de **backend**, para isso, utilizaremos a linguagem PHP, nossa preocupação não será o **frontend**, e sim entendermos o comportamento dessa linguagem dentro da programação Web, com o andamento do curso iremos refinando e melhorando os conceitos.

Mas antes vamos imaginar a internet na qual você pode apenas consumir conteúdos, como se fosse um jornal, uma revista, ou ainda, um programa na televisão. Chato, né? Mas quando se aprende as linguagens da web, como HTML e CSS, podemos montar sites que são como revistas e servem apenas para leitura, sem permitir interação com os internautas.

O segredo da famosa web 2.0 é a capacidade de interação entre as pessoas e os serviços online. Mas, para que esta interação seja possível, é necessário que os sites sejam capazes de receber informações dos internautas e também de exibir conteúdos personalizados para cada um, ou de mudar seu conteúdo automaticamente, sem que o desenvolvedor precise criar um novo HTML para isso.

Estes dois tipos de sites são chamados de estático e dinâmico, respectivamente.

Você já parou para pensar em tudo o que acontece quando você digita um endereço em seu navegador web? A história toda é mais ou menos assim:

- O navegador vai até o servidor que responde no endereço solicitado e pede a página solicitada.

- O servidor verifica se o endereço existe e se a página também existe em seu sistema de arquivos e então retorna o arquivo para o navegador.

- Após receber o arquivo HTML, o navegador começa o trabalho de renderização, para exibir a página para o usuário. É neste momento que o navegador também requisita arquivos de estilos (css), imagens e outros arquivos necessários para a exibição da página.

Quando se desenvolve páginas estáticas, este é basicamente todo o processo necessário para que o navegador exiba a página para o usuário. Chamamos de estáticas as páginas web que não mudam seu conteúdo, mesmo em uma nova requisição ao servidor.

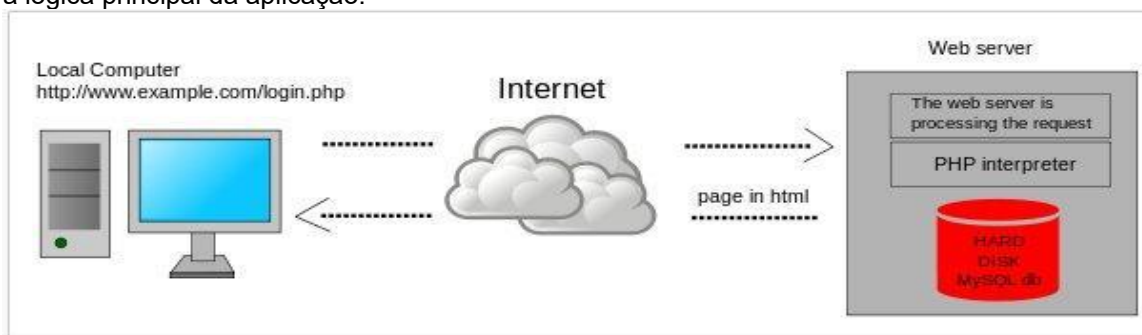
E como funciona uma página dinâmica?

O processo para páginas dinâmicas é muito parecido com o das páginas estáticas. A diferença é que a página será processada **no servidor** antes de ser enviada para o usuário. Este processamento no servidor é usado para alterar dinamicamente o conteúdo de uma página, seja ele HTML, CSS, imagens ou outros formatos.

Pense, por exemplo, em um site de um jornal. Em geral, este tipo de site contém algumas áreas destinadas às notícias de destaque, outras áreas para notícias gerais e ainda outras áreas para outros fins. Quando o navegador solicita a página para o servidor, ele irá montar o conteúdo antes de enviar para o navegador. Este conteúdo pode ser conseguido de algumas fontes, mas a mais comum é um banco de dados, onde, neste caso, as notícias ficam armazenadas para serem exibidas nas páginas quando necessário.

2. Scripting no lado do servidor (server-side)

A linguagem de servidor, ou *server-side scripting*, é a linguagem que vai rodar, fornecendo a lógica principal da aplicação.



Do lado do servidor significa toda a regra de negócio, acesso a banco de dados, a parte matemática e cálculos estarão armazenadas, enquanto que *local computer* será na máquina do usuário/cliente. Para a segurança e integridade dos dados, do lado do servidor geralmente é melhor, pois o usuário final não será capaz de ver ou manipular os dados que são enviados para o mesmo, e muito menos visualizar a regra de negócio, ou a forma de acesso ao banco de dados.

3. Scripting no lado do cliente (client-side)

As linguagens *client-side* são linguagens onde apenas o seu NAVEGADOR vai entender. Quem vai processar essa linguagem não é o servidor, mas o seu browser (Chrome, IE, Firefox,

etc...). Significa "lado do cliente", ou seja, aplicações que rodam no computador do usuário sem necessidade de processamento de seu servidor (ou host) para efetuar determinada tarefa.

Basicamente, ao se falar de aplicações client-side na web, estamos falando de *JavaScript*, e AJAX (*Asynchronous Javascript And XML*).

Existem vantagens e desvantagens ao utilizar o JavaScript e AJAX.

A principal vantagem está na possibilidade de você economizar bandwidth (largura de banda), e dar ao usuário uma resposta mais rápida de sua aplicação por não haver processamento externo.

Outra vantagem ao utilizar, agora o AJAX, seria o apelo visual de sua aplicação e rapidez de resposta. O que o AJAX faz é o processamento externo (server-side) parecendo ser interno (*client-side*). O usuário não percebe que houve um novo carregamento de página, pois ele busca informações no servidor e mostra rapidamente em um local específico da página através do JavaScript.

A principal desvantagem do *JavaScript* atualmente é que o usuário pode desativá-lo em seu navegador. Se a sua aplicação basear-se exclusivamente em JavaScript, nesse caso, ela simplesmente não vai funcionar.

4. A diferença entre o lado do cliente (*client-side*) e do lado do servidor (*server-side*) Scripting

Ao escrever aplicações para a web, você pode colocar os programas, ou scripts, ou no servidor da Web ou no navegador do cliente, a melhor abordagem combina uma mistura cuidadosa dos dois. Endereços de scripts do lado do servidor de gerenciamento e segurança de dados, enquanto que o script do lado do cliente se concentra principalmente na verificação de dados e layout de página.

Um servidor web é um computador separado e software com a sua própria conexão à Internet. Quando o navegador solicita uma página, o servidor recebe o seu pedido e envia o conteúdo do navegador. Um script de programa que executa no servidor web gera uma página com base na lógica do programa e envia para o navegador do usuário. O conteúdo pode ser texto e imagens padrão, ou pode incluir scripts do lado do cliente. Seu navegador executa os scripts do lado do cliente, o que pode animar imagens da página web, solicitar os dados do servidor ou executar outras tarefas.

Para um website ter uma sessão, onde você faz *login*, fazer compras e outros pedidos, o servidor precisa para identificar o computador. Milhares de usuários podem ser registrados em, ao mesmo tempo, o servidor tem de distingui-los. *Scripting* do lado do servidor mantém o controle da identidade de um usuário através de alguns mecanismos diferentes, tais como variáveis de sessão. Quando você fizer *login*, o script do servidor cria uma identificação de sessão exclusiva para você. O script pode armazenar informações em variáveis que duram o tempo que você ficar conectado.

Muitas páginas da web têm formulários para você preencher com seu nome, endereço e outras informações. Para certificar que os dados vão corretamente, os scripts de validação são capazes de verificar que as datas e códigos postais contêm apenas números e os estados têm certas combinações de duas letras. Este processo é mais eficaz quando o script é executado no lado do cliente. Caso contrário, o servidor tem que receber os dados, verificá-lo, e enviar-lhe uma mensagem de erro. Quando o navegador faz isso, você envia os dados de volta para o servidor somente uma vez.

Quando uma sessão de web envolve peneirar grandes quantidades de dados, um *script* do lado do servidor faz este trabalho melhor. Por exemplo, um banco pode ter um milhão de clientes. Quando você entrar, ele deve buscar o seu registro a partir deste arquivo grande. Ao invés de enviá-lo por toda a sua conexão de Internet para o seu navegador, o servidor web solicita informações de um servidor de dados perto dele. Além de aliviar a Internet do tráfego de dados desnecessários, isso também melhora a segurança.

Podemos encontrar uma maior variedade de linguagens de programação em servidores do que em navegadores. Os programadores fazem mais *scripting* do lado do cliente com a linguagem Javascript. No lado do servidor, você pode escrever em linguagens como PHP, VBScript ou ColdFusion. Enquanto alguns programadores escrever scripts do lado do cliente para executar fora do navegador, isso é arriscado, uma vez que pressupõe que o computador sabe que a linguagem.

5. Mantendo o histórico do código

Quando ingressamos no mercado de trabalho, veremos que sempre existirá uma pressão por entregas rápidas de novas funcionalidades nos sistemas que desenvolveremos. Mas, apesar da pressa, será necessário prestar atenção no que estamos fazendo, mesmo se a alteração for pequena.

Ao mexermos em um código existente é importante tomarmos cuidado para não quebrar o que já funciona. Por isso, queremos mexer o mínimo possível no código.

Temos medo de remover código obsoleto, não utilizado ou até mesmo comentado, mesmo que mantê-lo já nem faça sentido. Não é incomum no mercado vermos código funcional acompanhado de centenas de linhas de código comentado. Sem dúvida, é interessante manter o histórico do código dos projetos, para entendermos como chegamos até ali. Mas manter esse histórico junto ao código atual, com o decorrer do tempo, deixa nossos projetos confusos, poluídos com trechos e comentários que poderiam ser excluídos sem afetar o funcionamento do sistema. Seria bom se houvesse uma maneira de navegarmos pelo código do passado, como uma máquina do tempo para código.

5.1. Trabalhando em equipe

Mas, mesmo que tivéssemos essa máquina do tempo, temos outro problema: muito raramente trabalhamos sozinhos. Construir um sistema em equipe é um grande desafio, temos feito bastante práticas com isso em nossas aulas.

Nosso código tem que se integrar de maneira transparente e sem emendas com o código de todos os outros membros da nossa equipe. Como podemos detectar que estamos alterando o mesmo código que um colega? Como mesclar as alterações que fizemos com a demais alterações da equipe?

E como identificar conflitos entre essas alterações? Fazer isso manualmente, com cadernetas ou planilhas e muita conversa, parece trabalhoso demais e bastante suscetível a erros e esquecimentos. Seria bom que tivéssemos um robô de integração de código, que fizesse todo esse trabalho automaticamente.

5.2. Sistemas de controle de versão

Existem ferramentas que funcionam como máquinas do tempo e robôs de integração para o seu código. Elas nos permitem acompanhar as alterações desde as versões mais antigas.

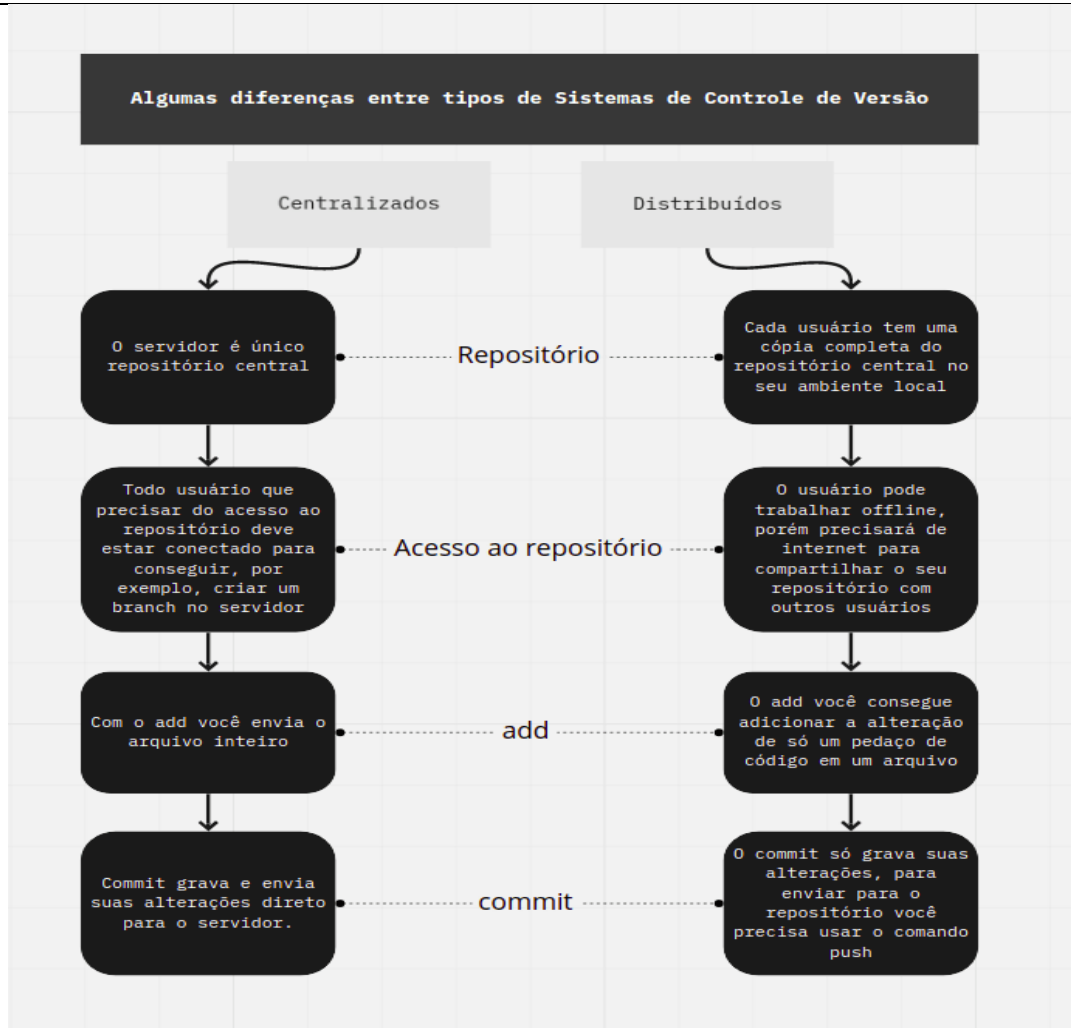
Também é possível detectar e mesclar alterações nos mesmos arquivos, além de identificar conflitos, tudo de maneira automática. Essas ferramentas são chamadas de sistemas de controle de versão. Nesse tipo de ferramenta, há um repositório que nos permite obter qualquer versão já existente do código. Sempre que quisermos controlar as versões de algum arquivo, temos que informar que queremos rastreá-lo no repositório. A cada mudança que desejamos efetivar, devemos armazenar as alterações nesse repositório.

Alterações nos mesmos arquivos são mescladas de maneira automática sempre que possível. Já possíveis conflitos são identificados a cada vez que obtemos as mudanças dos nossos colegas de time.

Desde a década de 1990, existe esse tipo de ferramenta. Alguns exemplos de sistemas de controle de versão mais antigos são CVS, *ClearCase*, *SourceSafe* e SVN (que ainda é bastante usado nas empresas). Em meados da década de 2000, surgiram sistemas de controle de versão mais modernos, mais rápidos e confiáveis, como Mercurial, Bazaar e, é claro, Git.

Precisamos entender que ele é o responsável por gravar todas as alterações feitas, e além disso, precisa permitir que todas as alterações feitas ocorram paralelamente e não comprometam o andamento do projeto.

Os sistemas de versionamento podem ser classificados como distribuídos ou centralizados, vamos ver abaixo:



A partir dessa imagem, conseguimos ver algumas diferenças e o que chama mais atenção — é importante focar sem entrar tanto em partes técnicas — é o que mais diferencia: a forma como trabalhamos com eles e também o ambiente local que o Git nos proporciona, dando uma cópia completa do repositório. Apesar de alguns fluxos terem o mesmo nome, eles funcionam de formas distintas.

5.3. O que é o Git?

Como sempre falamos, precisamos entender a história e conceito do que vamos estudar para termos plena ciência do conceito que iremos aplicar.

O Git é um sistema de controle de versão distribuído, o que significa que um clone local do projeto é um repositório de controle de versão completo. Esses repositórios locais totalmente funcionais facilitam o trabalho offline ou remoto. Os desenvolvedores confirmam o trabalho localmente e depois sincronizam a cópia do repositório com a cópia no servidor. Esse paradigma é diferente do controle de versão centralizado, no qual os clientes devem sincronizar o código com um servidor antes de criar novas versões do código.

A flexibilidade e a popularidade do Git o tornam uma ótima escolha para qualquer equipe. Muitos desenvolvedores e universitários já sabem como usar o Git. A comunidade de usuários do Git criou recursos para treinar desenvolvedores, e a popularidade do Git facilita a obtenção de ajuda quando necessário. Quase todos os ambientes de desenvolvimento têm suporte ao Git e as ferramentas de linha de comando do Git implementadas em todos os principais sistemas operacionais.

Toda a história da criação do *Git* se dá em torno do *Kernel* do Linux. Ele e seu versionamento foram mantidos durante muito tempo no *Bitkeeper*, mas a empresa por trás dele, a *BitMoover*, resolveu ir para uma linha de comercialização do licenciamento de acesso ao *software*.

Por um tempo, ela permitiu que os desenvolvedores do kernel do Linux usassem a versão gratuita (*Freeware*), mas um dos mantenedores, Andrew Tridgell, desenvolveu uma versão *open source* do *client*, uma ferramenta que poderia trabalhar com o código fonte armazenado no *BitKeeper*. Porém, a *BitMoover* ficou extremamente insatisfeita e considerou como violação da licença e engenharia reversa a criação desse *client*.

Após meses de negociação, esse acesso gratuito foi revogado e surgiu a necessidade da criação de algo que não só substituísse. Mas fosse melhor que o *Bitkeeper*. Linus e todas as pessoas que mantinham o Kernel começaram a busca, mas nenhuma das opções de controle de versão descentralizadas disponíveis atendiam às necessidades.

Nasce, então, o *Git* com a proposta de resolver a necessidade de um sistema de distribuição descentralizada, e que segue a filosofia de software livre, assim como o Linux, e sem se preocupar em ficar preso a uma ferramenta proprietária que de um dia pro outro os criadores resolvessem comercializá-lo.

Visto de um ângulo técnico, o *Git* é uma ferramenta semelhante ao *Bitkeeper* por trabalhar de forma distribuída e uma coisa que acho interessante é que grande parte do *flow* que conhecemos no *Git*, veio do *flow* aprendido com o uso do *Bitkeeper*. Porém, mesmo com essa semelhança a ideia nunca foi ter um clone.

Quando foi criado, a pretensão era resolver exclusivamente o problema relacionado ao Kernel do Linux. Mas hoje o *Git* se tornou uma das ferramentas mais utilizadas no mundo inteiro para controle de versão.

5.4. Benefícios do Git

5.4.1. Desenvolvimento simultâneo:

Todos têm sua própria cópia local do código e podem trabalhar simultaneamente em seus próprios *branches*. O *Git* funciona offline, já que quase todas as operações são locais.

5.4.2. Lançamentos mais rápidos

Os *branches* permitem o desenvolvimento flexível e simultâneo. O *branch* principal contém um código estável e de alta qualidade, com base no qual você lançará versões. Os *branches* de recurso contêm trabalhos em andamento, que são mesclados no *branch* principal após a conclusão. Ao separar o *branch* de lançamento do desenvolvimento em andamento, é mais fácil gerenciar um código estável e enviar atualizações mais rapidamente.

5.4.3. Integração interna

Devido à sua popularidade, o *Git* pode ser integrado à maioria dos produtos e ferramentas. Todas as principais IDEs têm suporte interno ao *Git*, e muitas ferramentas oferecem suporte à integração contínua, implantação contínua, testes automatizados, rastreamento de itens de trabalho, métricas e integração de recursos de relatório com o *Git*. Essa integração simplifica o fluxo de trabalho diário.

5.4.4. Forte suporte da comunidade

O *Git* usa um código aberto e se tornou o padrão de fato para controle de versão. Há inúmeras opções de ferramentas e recursos disponíveis para as equipes aproveitarem. O volume de suporte da comunidade para o *Git* em comparação com outros sistemas de controle de versão facilita a obtenção de ajuda quando necessário.

5.4.5. O *Git* funciona com qualquer equipe

Usar o *Git* com uma ferramenta de gerenciamento de código-fonte aumenta a produtividade de uma equipe, incentivando a colaboração, aplicando políticas,

automatizando processos e melhorando a visibilidade e a rastreabilidade do trabalho. A equipe pode se contentar com ferramentas individuais para controle de versão, rastreamento de itens de trabalho e integração e implantação contínuas. Ou os membros da equipe podem escolher uma solução como o GitHub ou o Azure DevOps que ofereça suporte a todas essas tarefas em um só lugar.

5.4.6. Solicitações *pull*

Use solicitações *pull* para discutir alterações de código com a equipe antes de mesclá-las no branch principal. As discussões em solicitações *pull* são inestimáveis para garantir a qualidade do código e melhorar o conhecimento de toda a sua equipe. Plataformas como o GitHub e o Azure DevOps oferecem uma experiência avançada de solicitação de *pull*, em que os desenvolvedores podem procurar alterações de arquivo, deixar comentários, inspecionar commits, exibir compilações e votar para aprovar o código.

5.4.7. Políticas de *branch*

As equipes podem configurar o GitHub e o Azure DevOps para aplicar fluxos de trabalho e processos consistentes em toda a equipe. Eles podem configurar políticas de *branch* para garantir que as solicitações *pull* atendam aos requisitos antes da conclusão. As políticas de *branch* protegem *branches* importantes, evitando transmissões diretas, exigindo revisores e garantindo compilações limpas.

5.5. O que é o GitHub?

GitHub é uma plataforma de hospedagem de códigos-fonte e de outros tipos de arquivos que utiliza o *software* Git. Como a GitHub é uma plataforma baseada na nuvem e está disponível na Web, o Git assume o papel de Sistema de Controle de Versões distribuído.

Dentro do GitHub, os projetos podem ter tarefas atribuídas a pessoas específicas ou a equipes, com definições de permissões e funções por login - como a função de moderador do projeto, por exemplo.

Para quem é do mundo do desenvolvimento de códigos, o GitHub é uma boa oportunidade para aprimorar conhecimentos, compartilhar projetos e construir portfólio, muitas empresas a utilizam para realização de testes em processos seletivos, e consequentemente consulta para ver os projetos de seu candidato.

O GitHub é uma plataforma que contém uma versão gratuita e planos privados com funcionalidades adicionais. Para saber se a versão gratuita atende sua necessidade é preciso fazer um cadastro e testar. A pessoa não precisa ter conhecimentos em programação – afinal, o GitHub admite diversos tipos de arquivo. Até por isso, sua interface é muito amigável.

5.5.1. As principais funcionalidades da ferramenta são:

5.5.1.1. Repositório

O repositório é o local onde são armazenados os arquivos do projeto. Esse diretório pode ficar na GitHub ou em um dispositivo local. Ele admite arquivos de vários formatos, como imagem, áudio, o código-fonte, textos e qualquer outro tipo de documento necessário ao projeto.

5.5.1.2 *Branches*

Esse recurso é também chamado de “ramificações” e fica dentro do repositório. Aqui, a pessoa consegue promover alterações isoladamente, sem afetar a versão original, e pode ter uma visão de como as mudanças propostas se comportam, ao mesmo tempo em que pode comparar as versões.

As *branches* permitem desenvolver funcionalidades, consertar bugs e experimentar ideias de maneira segura.

5.5.1.3. Commits

Cada alteração promovida em um arquivo precisa ser executada e salva para se criar outra branch. O nome dessa ação é *Commit*.

5.5.1.4. Pull Requests

Quando profissionais de desenvolvimento chegam a uma versão considerada boa o suficiente, é possível criar um *pull request* para sugerir a implementação ao original. Nisso, as outras pessoas envolvidas no projeto vão poder analisar essa colaboração, podendo incorporá-las ou não.

As pessoas do projeto comparam o original e o que está na *branch* do *pull request*. Se for aprovada, é possível abrir uma solicitação de mesclagem (ou merge) das mudanças ao projeto original.

Para quem deseja ter uma visão mais abrangente de tudo o que o GitHub disponibiliza, recomendamos dar uma olhada no GitHub Docs.

Os impressionantes números do GitHub

- O GitHub é utilizado por mais de 3 milhões de profissionais de desenvolvimento brasileiros, sendo o terceiro maior público em número de logins.
- Em 2018, o GitHub foi comprado pela Microsoft pela cifra de US\$ 7,5 bilhões.
- O GitHub conta com mais de 100 milhões profissionais de desenvolvimento, mais de 4 milhões de organizações cadastradas e mais de 420 milhões de repositórios.
- Mais de 37 milhões de negócios já adotaram o Copilot GitHub. Desenvolvida pela OpenAI, o Copilot GitHub tem como objetivo de criar linhas de código automáticos ou por meio da solicitação de quem usa.
- O recurso dá suporte a milhares de linguagens, entre elas *Python*, *JavaScript*, *TypeScript*, *Ruby*, *Java* e *Go*, sendo bastante útil para resolver dúvidas ou criar soluções novas.

5.6. Git no Windows

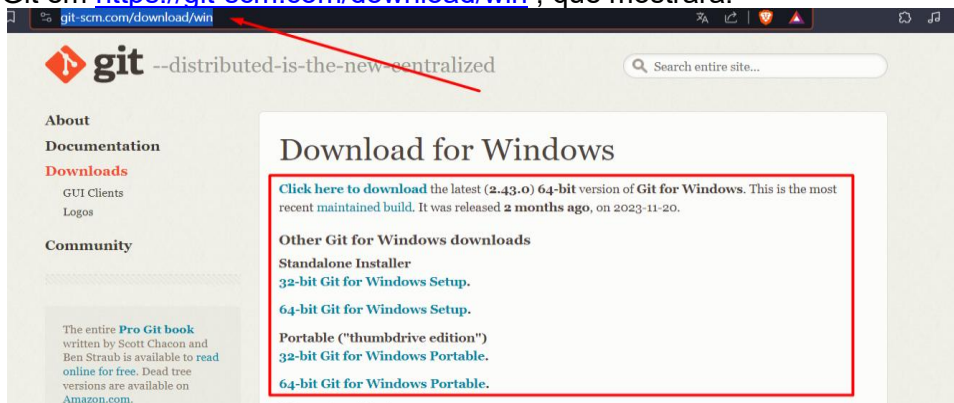
Podemos instalar o Git em outros sistemas operacionais, em nosso caso iremos trabalhar com o Windows, por isso, mostrarei um passo a passo para o mesmo, e também como podemos manipular o mesmo.

5.6.1. Ambiente para manuseio do Git

Temos duas formas de manipular o Git no *Windows*, através de IDE ou através de linha de comando, como estamos com foco em desenvolvimento de software a ideia aqui é utilizarmos prompt de comando, pois o mercado é muito acostumado com essa prática. Para *prompt* de comando, podemos utilizar o cmd do Windows, terminal que está embutido no *Visual Studio Code*, *PowerShell*, o *Cmder* também é muito utilizado.

5.6.2. Instalando o Git no Windows

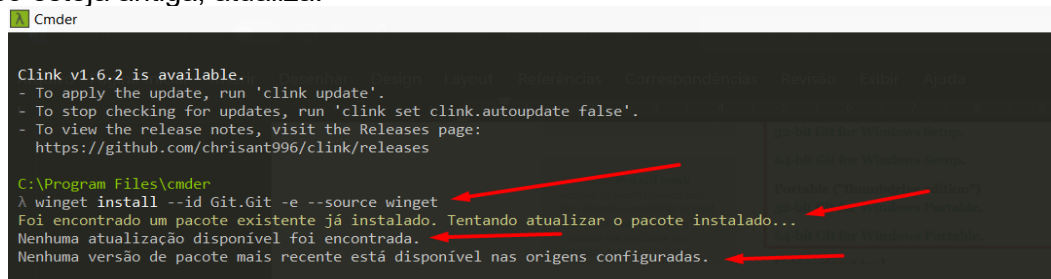
Temos duas formas de fazer a instalação, através do download diretamente no site do Git em <https://git-scm.com/download/win>, que mostrará:



Ou através de comando (winget - Windows Package Manager) que colocamos no prompt de comando escolhido:

```
winget install --id Git.Git -e --source winget
```

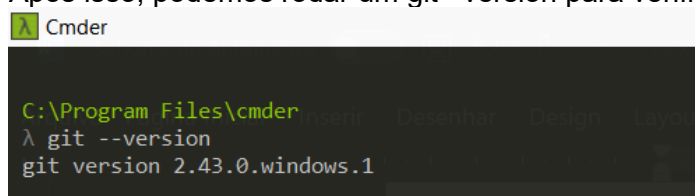
Caso tenha uma versão do Git instalada em seu computador, ele verifica a versão, e caso esteja antiga, atualiza:



```
C:\Program Files\cmdr
Clink v1.6.2 is available.
- To apply the update, run 'clink update'.
- To stop checking for updates, run 'clink set clink.autoupdate false'.
- To view the release notes, visit the Releases page:
  https://github.com/chrisant996/clink/releases

C:\Program Files\cmdr
λ winget install --id Git.Git -e --source winget
Foi encontrado um pacote existente já instalado. Tentando atualizar o pacote instalado...
Nenhuma atualização disponível foi encontrada.
Nenhuma versão de pacote mais recente está disponível nas origens configuradas.
```

Após isso, podemos rodar um `git --version` para verificar:



```
C:\Program Files\cmdr
λ git --version
git version 2.43.0.windows.1
```

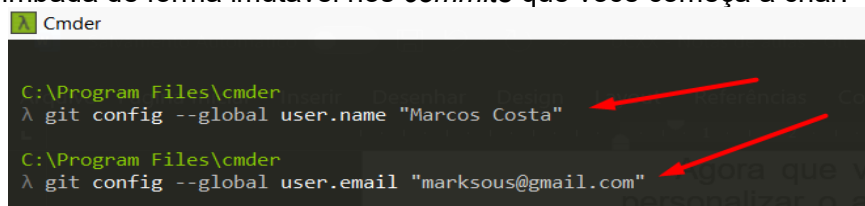
5.6.3. Configuração inicial do Git

Agora que você tem o Git em seu sistema, podemos fazer algumas coisas para personalizar o ambiente Git. Faremos apenas uma vez por computador e o efeito se manterá após atualizações. Mas também podemos mudá-las em qualquer momento rodando esses comandos novamente.

O Git vem com uma ferramenta chamada **git config** que permite ver e atribuir variáveis de configuração que controlam todos os aspectos de como o Git aparece e opera. No Windows, Git procura pelo arquivo `.gitconfig` no diretório `$HOME` (`C:\Users\$USER` para a maioria). Ele também procura por `/etc/gitconfig`, mesmo sendo relativo à raiz do sistema, que é onde quer que você tenha instalado Git no seu sistema.

5.6.4. Sua Identidade

A primeira coisa que devemos fazer ao instalar Git é configurar nosso nome de usuário e endereço de e-mail. Isto é importante porque cada *commit* usa esta informação, e ela é carimbada de forma imutável nos *commits* que você começa a criar:



```
C:\Program Files\cmdr
λ git config --global user.name "Marcos Costa"


C:\Program Files\cmdr
λ git config --global user.email "marksous@gmail.com"
```

Reiterando, você precisará fazer isso somente uma vez se tiver usado a opção `--global`, porque então o Git usará esta informação para qualquer coisa que você fizer naquele sistema. Se você quiser substituir essa informação com nome diferente para um projeto específico, você pode rodar o comando sem a opção `--global` dentro daquele projeto.

5.6.5. Seu Editor

Agora que a sua identidade está configurada, você pode escolher o editor de texto padrão que será chamado quando Git precisar que você entre uma mensagem. Se não

for configurado, o Git usará o editor padrão, que normalmente é o Vim. Em nosso caso, iremos configurar o *Visual Studio Code*, você pode fazer o seguinte:

 Cmder

```
C:\Program Files\cmdr
λ git config --global core.editor "code --wait"
```

5.6.6. Iniciando um repositório no Git

Para começarmos os trabalhos, devemos criar um repositório local de seu projeto, para isso, pense em seu computador, em um local onde ele ficará armazenado.

Para exemplificar, irei criar um repositório dentro da pasta htdocs do XAMPP, nesse caso para programas desenvolvidos com PHP, para isso utilizo o comando CD para “navegar” nas pastas em meu computador.

 Cmder

```
C:\Program Files\cmdr
λ cd\
C:\
λ cd xampp
C:\xampp
λ cd htdocs
```

Após isso, eu crio a pasta que representará meu repositório, o comando **mkdir** serve para criarmos a pasta, o **ls** serve para vermos o conteúdo do disco.

 Cmder

```
C:\xampp\htdocs
λ mkdir aula-git
C:\xampp\htdocs
λ ls
applications.html  aula-git/  compras/  estagio/  estagio_marcos/  estagio_old2809/
aula.php          bitnami.css  esp32/    estagio.zip  estagio_old/    estagio-20231009T142010Z-001.zip
```

Podemos observar que a pasta aula-git foi criada.

```
C:\xampp\htdocs
λ ls
applications.html  aula-git/  compras/  estagio/  estagio_marcos/
aula.php          bitnami.css  esp32/    estagio.zip  estagio_old/
```

Nesse momento nosso repositório foi criado, mas ainda não foi iniciado para isso precisamos entrar no repositório criado:

 Cmder

```
C:\xampp\htdocs
λ cd aula-git
C:\xampp\htdocs\aula-git
λ |
```

Agora iniciamos o repositório com o **git init**.

```
Cmder
C:\xampp\htdocs> cd aula-git
C:\xampp\htdocs\aula-git> git init
Initialized empty Git repository in C:/xampp/htdocs/aula-git/.git/
C:\xampp\htdocs\aula-git (master)>
```

Observem que o Git iniciou o repositório na *branch master*, iremos falar disso mais tarde, mas vamos nos atentar que ele coloca uma pasta *.git*, se irmos até mesma teremos:

```
C:\xampp\htdocs\aula-git (master)> cd .git
C:\xampp\htdocs\aula-git\.git (master)> ls -la
total 11
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 ./
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 ../
-rw-r--r-- 1 marco 197610 130 jan 29 12:36 config
-rw-r--r-- 1 marco 197610  73 jan 29 12:36 description
-rw-r--r-- 1 marco 197610  23 jan 29 12:36 HEAD
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 hooks/
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 info/
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 objects/
drwxr-xr-x 1 marco 197610  0 jan 29 12:36 refs/
```

Nesta pasta temos arquivos e pastas sobre a configuração necessária para que o Git trabalhe em nosso projeto.

Agora temos o repositório de nosso projeto criado e com o Git iniciado.

5.6.7. Criando arquivo e verificando a pasta

Agora vamos fazer um simples exemplo de uma página html para simularmos um projeto, vamos abrir o *Visual Studio Code* com o arquivo **index.html**, vejamos:

```
Cmder
C:\xampp\htdocs\aula-git (master)> code index.html
```

Com o VSCode aberto, iremos fazer uma simples página.

```
File Edit Selection View Go Run ...
index.html
C:\> xampp > htdocs > aula-git > index.html > ...
1  <html>
2    <head>
3      <title>Aula de Git</title>
4    </head>
5    <body>
6      <h1>Criando um arquivo.</h1>
7    </body>
8  </html>
```

Ao voltarmos ao prompt, com o comando **ls**, verificamos que o arquivo foi salvo corretamente.

```
C:\xampp\htdocs\aula-git (master)
λ ls
index.html
```

Agora temos que entender que o nosso arquivo está vinculado ao nosso computador, para mandarmos para um repositório na nuvem, devemos ter uma conta e o repositório criado por exemplo, no GitHub, com isso temos que vincular nosso repositório local com o remoto:

```
C:\xampp\htdocs\aula-git (master)
λ git remote add origin https://github.com/marccossousa33dev/aula_git.git
```

Apesar de já termos criado o arquivo index.html, vamos criar a branch para trabalharmos nosso arquivo, para isso vamos criá-la e já mudarmos para a mesma.

```
C:\xampp\htdocs\aula-git (master)
λ git checkout -b exemploAula
Switched to a new branch 'exemploAula'

C:\xampp\htdocs\aula-git (exemploAula)
λ
```

Agora vamos verificar o status desta Branch.

```
C:\xampp\htdocs\aula-git (exemploAula)
λ git status
On branch exemploAula

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    index.html

nothing added to commit but untracked files present (use "git add" to track)

C:\xampp\htdocs\aula-git (exemploAula)
λ
```

Somos informado que temos um arquivo que não está commitado e para que isso aconteça, ele informa que temos que usar o git add . ou git add index.html.

```
C:\xampp\htdocs\aula-git (exemploAula)
λ git add .

C:\xampp\htdocs\aula-git (exemploAula)
λ
```

Se repetirmos o comando git status, temos:

```
C:\xampp\htdocs\aula-git (exemploAula)
λ git status
On branch exemploAula

No commits yet

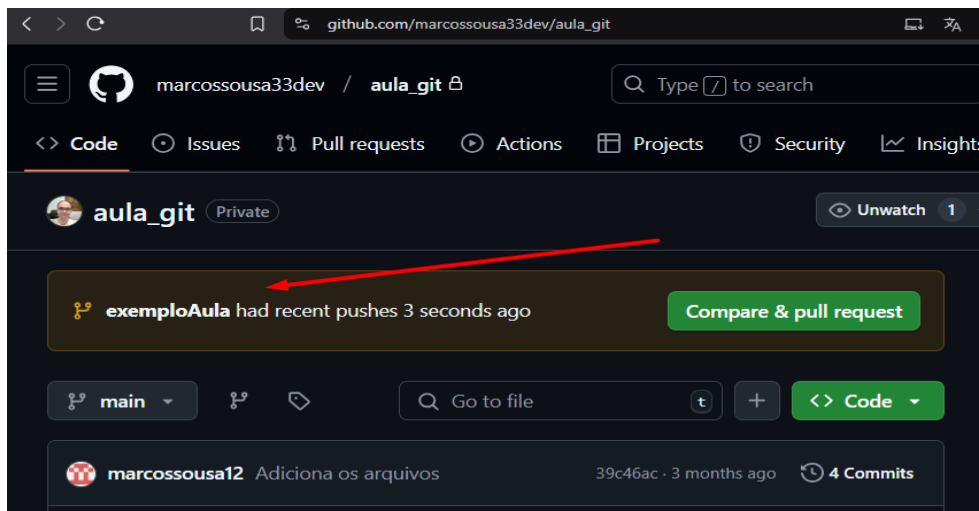
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html
```

Temos um arquivo para ser commitado.

```
C:\xampp\htdocs\aula-git (exemploAula)
λ git commit -m "Aplicando exemplo de commit"
[exemploAula (root-commit) 6b2cca7] Aplicando exemplo de commit
1 file changed, 8 insertions(+)
create mode 100644 index.html
```

Neste momento nosso arquivo está pronto para ser enviado ao repositório remoto

```
C:\xampp\htdocs\aula-git (exemploAula)
λ git push git@github.com:marcossousa33dev/aula_git.git
Enter passphrase for key '/c/Users/marco/.ssh/id_rsa':
Enter passphrase for key '/c/Users/marco/.ssh/id_rsa':
Enter passphrase for key '/c/Users/marco/.ssh/id_ed25519':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 315 bytes | 315.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'exemploAula' on GitHub by visiting:
remote:   https://github.com/marcossousa33dev/aula_git/pull/new/exemploAula
remote:
To github.com:marcossousa33dev/aula_git.git
 * [new branch]      exemploAula -> exemploAula
```



Nosso arquivo agora está no repositório remoto, agora só precisamos fazer o Pull Request e colocar na main.

Resumindo, abaixo os principais comandos, lembrando que devemos praticar para fixarmos melhor o conteúdo:

Configuração Inicial

```
git config --global user.name "Seu Nome"
git config --global user.email "seuemail@example.com"
```

Criar um Repositório

```
git init
```

Clonar um Repositório

```
git clone <url-do-repositorio>
```

Verificar o Status do Repositório

```
git status
```

Adicionar Arquivos ao Índice (Staging Area)

```
git add <arquivo>
git add .
```

Fazer um Commit

```
git commit -m "Mensagem do commit"
```

Verificar o Histórico de Commits

```
git log
```

Enviar Alterações para o Repositório Remoto


```
git push origin <nome-da-branch>
```

Atualizar o Repositório Local com Alterações do Remoto

```
git pull
```

Criar uma Nova Branch

```
git checkout -b <nome-da-branch>
```

Trocar para uma Branch Existente

```
git checkout <nome-da-branch>
```

Verificar Branches Existentes

```
git Branch
```

Excluir uma Branch

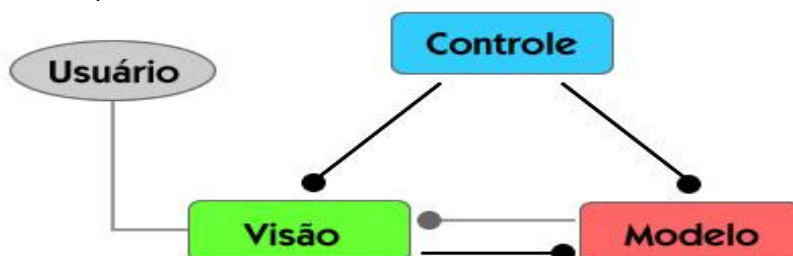
```
git branch -d <nome-da-branch>
```

5. Padrão MVC

MVC é o acrônimo de *Model-View-Controller* (em português, Modelo-Visão-Controle) e que, como o próprio nome supõe, separa as camadas de uma aplicação em diferentes níveis. Ao contrário do que muitos desenvolvedores dizem, o MVC não é um Padrão de Projeto, mas sim um Padrão de Arquitetura de Software.

O MVC define a divisão de uma aplicação em três componentes: Modelo, Visão e Controle. Cada um destes componentes tem uma função específica e estão conectados entre si. O objetivo é separar a arquitetura do software para facilitar a compreensão e a manutenção.

A camada de Visão é relacionada ao visual da aplicação, ou seja, as telas que serão exibidas para o usuário. Nessa camada apenas os recursos visuais devem ser implementados, como janelas, botões e mensagens. Já a camada de Controle atua como intermediária entre as regras de negócio (camada Modelo) e a Visão, realizando o processamento de dados informados pelo usuário e repassando-os para as outras camadas. E por fim, temos a camada de Modelo, que consiste na essência das regras de negócio, envolvendo as classes do sistema e o acesso aos dados. Observe que cada camada tem uma tarefa distinta dentro do software.

**5.1 Explicação do conceito do MVC****5.1.1 Certo, mas qual é a vantagem?**

A princípio, pode-se dizer que o sistema fica mais organizado quando está estruturado com MVC. Um novo desenvolvedor que começar a trabalhar no projeto não terá grandes dificuldades em compreender a estrutura do código. Além disso, as exceções são mais fáceis de serem identificadas e tratadas. Ao invés de revirar o código atrás do erro, o MVC pode indicar em qual camada o erro ocorre. Outro ponto importante do MVC é a segurança da transição de dados. Através da camada de Controle, é possível evitar que qualquer dado inconsistente chegue à camada de Modelo para persistir no banco de dados. Imagine a camada de Controle como uma espécie de Firewall: o usuário entra com os dados e a camada de Controle se responsabiliza por bloquear os dados que venham a causar inconsistências no banco de dados. Portanto, é correto afirmar que essa camada é muito importante.

5.1.2 Posso ter apenas três camadas no MVC?

Eis que essa é outra dúvida bastante questionada pelos desenvolvedores. Embora o MVC sugira a organização do sistema em três camadas, nada impede que você “estenda” essas camadas para expandir a dimensão do projeto. Por exemplo, vamos supor que um determinado

projeto tenha um módulo web para consulta de dados. Aonde esse módulo se encaixaria dentro das três camadas?

Bem, este módulo poderia ser agrupado com os outros objetos da camada Visão, mas seria bem mais conveniente criar uma camada exclusiva (como “Web”) estendida da camada de Visão e agrupar todos os itens relacionados a esse módulo dentro dela. O mesmo funcionaria para um módulo Mobile ou Webservice. Neste caso, a nível de abstração, essas novas camadas estariam dentro da camada de Visão, mas são unidades estendidas.

5.1.3 Então podemos concluir resumidamente

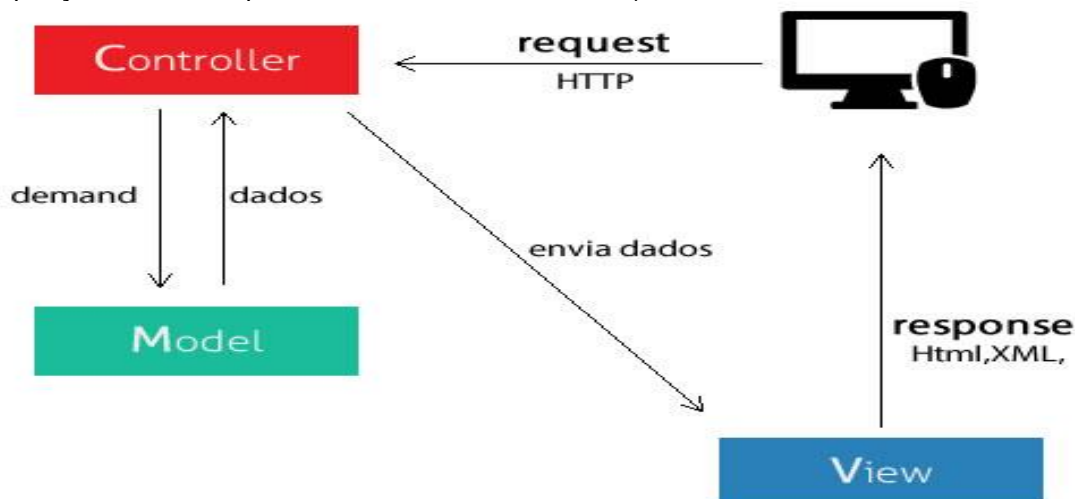
MVC é nada mais que um padrão de arquitetura de software, separando sua aplicação em 3 camadas. A camada de interação do usuário (*view*), a camada de manipulação dos dados(*model*) e a camada de controle(*controller*).

Model: quando você pensar em manipulação de dados, pense em model. Ele é responsável pela leitura e escrita de dados, e também de suas validações.

View: é a camada de interação com o usuário. Ela apenas faz a exibição dos dados.

Controller: é o responsável por receber todas as requisições do usuário. Seus métodos chamados *actions* são responsáveis por uma página, controlando qual *model* usar e qual *view* será mostrado ao usuário.

(A imagem a seguir representa o fluxo do MVC em um contexto de Internet, com uma requisição HTTP e resposta em formato HTML ou XML)



5.1.4 O diálogo das camadas na Web (Brincadeirainha)

View - Fala **Controller**! O usuário acabou de pedir para acessar o Facebook! Pega os dados de *login* dele aí.

Controller – Blz. Já te mando a resposta. Aí *model*, meu parceiro, toma esses dados de *login* e verifica se ele loga.

Model – Os dados são válidos. Mandando a resposta de login.

Controller – Blz. **View**, o usuário informou os dados corretos. Vou mandar pra vc os dados dele e você carrega a página de perfil.

View – Vlw. Mostrando ao usuário...

5.2 API: conceito, exemplos de uso e importância da integração para desenvolvedores

A integração a APIs pode facilitar demais o trabalho de quem trabalha com desenvolvimento. Entenda melhor o conceito de API e veja exemplos de como elas funcionam.

A sigla API é uma abreviação para *Application Programming Interface*, ou, em português, interface de programação de aplicação.

5.2.1 API: Conceito

Em uma definição formal, o conceito de API está relacionado a um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por outros aplicativos.

O conceito de API nada mais é do que uma forma de comunicação entre sistemas. Elas permitem a integração entre dois sistemas, em que um deles fornece informações e serviços que podem ser utilizados pelo outro, sem a necessidade de o sistema que consome a API conhecer detalhes de implementação do software.

Metaforicamente, podemos pensar no que é API como um garçom. Quando estamos em um restaurante, buscamos o que desejamos no menu e solicitamos ao garçom. O garçom encaminha esse pedido à cozinha, que prepara o pedido. No fim, o garçom traz o prato pronto até a gente. Não temos detalhes de como esse prato foi preparado, apenas recebemos o que solicitamos.

Com os exemplos de API disponíveis, a ideia é a mesma do garçom. Ela vai receber seu pedido, levar até o sistema responsável pelo tratamento e te devolver o que solicitou (o que pode ser uma informação, ou o resultado do processamento de alguma tarefa, por exemplo).

5.2.2 A API de integração são grandes aliadas dos *devs* e das empresas.

Importância das APIs:

Atualmente, quando temos um mundo tão conectado, as APIs são essenciais para a entrega de produtos cada vez mais ricos para os usuários.

Independente do produto que se deseja oferecer, seja ele um site, um aplicativo, um *bot*, é muito importante a utilização do conceito de APIs integradas a sistemas para trazer uma gama de funcionalidades para sua aplicação.

Podemos pensar na utilidade do que é API por dois pontos de vista: como produtor ou consumidor.

Quando você produz, você está criando APIs para que outras aplicações ou sistemas possam se integrar ao seu sistema. Isso não significa que você irá criar uma API apenas para expor seu sistema a terceiros. Vai depender do seu propósito, mas a API também pode ser apenas para seu uso, como de uma interface sua para os clientes.

Seguindo nessa linha de raciocínio, imagine que você possua um sistema de comércio. Ao criar uma API de acesso ao seu sistema, você possibilita a oferta de seus produtos nas interfaces de seu interesse. O que você vai expor de seu sistema na API depende do que deseja oferecer como funcionalidade, mas poderia, por exemplo:

- Listar produtos;
- Ofertar promoções;
- Efetivar vendas;
- Realizar cobrança do pedido.

Algumas aplicações do conceito de API

Até aqui, exploramos o que é API e as possibilidades de você criar a API para facilitar a integração com aplicações.

No entanto, hoje em dia já possuímos uma quantidade enormes de APIs que podemos utilizar para enriquecer nossas aplicações, e são de todos os tipos. Vamos citar um exemplo de verificação de CEP.

Em nosso exemplo vamos utilizar o ViaCep que é um webservice que serve para consulta de endereços no Brasil, ele é gratuito, para consultar o cep basta fazermos uma requisição *http* para a API do ViaCep, e então obter o retorno com informações como CEP, nome da cidade, código do município, UF e mais.

Para verificar como fazemos essa parte, o fornecedora da API pode disponibilizar a documentação da mesma, nesse caso acessando a url: <https://viacep.com.br/> nos é relatado como fazermos a consulta dos dados.

Para esse exemplo e vermos o retorno dos dados, vamos utilizar o *INSOMNIA* (<https://insomnia.rest/>) para testarmos a requisição, que é uma ferramenta popular de *API Client* usada para testar, enviar e depurar requisições HTTP e APIs REST, e que também utilizaremos em nossas aulas, que serve para testarmos através de métodos *POST* nossos *ENDPOINTS* para nossas APIs.

Passos:

1º - Criamos um nome para nossa requisição;

2º - Passamos a URL da API de acordo com a documentação vista no ViaCep.

Validação do CEP

Quando consultado um CEP de formato inválido, por exemplo: "950100100" (9 dígitos), "95010A10" (alfanumérico), "95010 10" (espaço), o código de retorno da consulta será um 400 (Bad Request). Antes de acessar o webservice, valide o formato do CEP e certifique-se que o mesmo possua {8} dígitos. Exemplo de como validar o formato do CEP em javascript está disponível nos exemplos abaixo.

Quando consultado um CEP de formato válido, porém inexistente, por exemplo: "99999999", o retorno conterá um valor de "erro" igual a "true". Isso significa que o CEP consultado não foi encontrado na base de dados. Veja como manipular este "erro" em javascript nos exemplos abaixo.

Formatos de Retorno

Veja exemplos de acesso ao webservice e os diferentes tipos de retorno:

JSON

URL: viacep.com.br/ws/01001000/json/
 UNICODE: viacep.com.br/ws/01001000/json/unicode/

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
}
```

3º - Enviamos a requisição;

4º - Retorno da API no formato JSON

Aula_PW_II GET <https://viacep.com.br/ws/06756030/json/> Send 200 OK 424 ms 252 B

desenv Cookies Body Auth Query Header Docs Preview Header 14 Cookie Timeline

Filter

Testes para aula

GET API CEP

Aula_Etec

POST API Login

```
1 {
2   "cep": "06756-030",
3   "logradouro": "Rua Marino Martins de Oliveira",
4   "complemento": "",
5   "bairro": "Parque Monte Alegre",
6   "localidade": "Taboão da Serra",
7   "uf": "SP",
8   "ibge": "3552809",
9   "gia": "6750",
10  "ddd": "11",
11  "siafi": "7157"
12 }
```

5.2.3 API REST

Representational State Transfer (REST), ou Transferência de Estado Representacional. Lá vem a Ciência da Computação, com suas siglas e nomes complexos.

Vamos simplificar um pouco. *API REST* é oriundo de uma dissertação de doutorado de Roy Fielding, nos anos 2000.

De maneira bem resumida, ele orienta a usar o protocolo HTTP para representar o estado/operação sobre a informação.

Em suma, o que precisamos saber é *API REST* é um serviço sem estado (*cookies* ou *session*) que utiliza corretamente os verbos HTTP:

- **POST**: para criar um objeto no servidor
- **DELETE**: para apagar um objeto do servidor.
- **PUT**: para atualizar um objeto no servidor.
- **GET**: para obter um ou mais objetos do servidor.

Ou seja, uma série de instruções que permitem a criação de um serviço com uma interface bem definida. Para isso, há uma série de recomendações com relação a rotas das *URLs*, dentre outras coisas.

Quando os princípios listados nessa recomendação são adotados em sua totalidade, temos uma *API RESTful*.

5.3 JSON

Vamos trabalhar em nosso projeto a passagem e retorno dos dados pelas *controllers* em formato *JSON*, mas afinal... o que é isso?

JSON é um formato de representação de dados baseado na linguagem de programação *JavaScript*, daí o nome *JavaScript Object Notation*. Ou seja, Notação de Objeto em Javascript.

Vamos pensar no exemplo de um objeto pessoa com nome Pedro e altura 1,90. A representação deste objeto em *JSON* ficaria assim:

```
{  
  "nome": "Pedro",  
  "altura": 1.90  
}
```

Este é um exemplo bem simples, mas podemos observar que o *JSON* não vai muito além disso.

O mais importante que você precisa saber sobre o *JSON* é que ele é composto por chave/valor (ou no inglês *key/value*), as chaves representam os nomes dos atributos da classe e os valores, bem, são os valores do objeto.

No exemplo acima, temos as chaves nome e altura e os valores Pedro e 1.90, respectivamente.

5.3.1 A sintaxe básica do JSON

A sintaxe de um *JSON* é bem simples como já falamos, mas agora vamos aprofundar um pouco mais.

Vejamos os elementos básicos do *JSON*.

{ e } - delimita um objeto.

[e] - delimita um *array*.

: - separa chaves (atributos) de valores.

, - separa os atributos chave/valor.

Entendendo estes quatro elementos básicos você já será capaz de ler um *JSON* com mais facilidade.

A leitura simples de um *JSON* é que ele representa um objeto que tem atributos e cada atributo tem valores.

Os *JSONs* são estruturados em objetos e/ou *arrays* (ou listas). Os objetos são representados por chaves { } e os *arrays* são representados por colchetes [].

Essa é a primeira coisa que você deve olhar no *JSON*: Onde estão as chaves e os colchetes?

Identificando estes dois tipos de estruturas você já sabe se os dados serão acessados por chaves (quando for um objeto), ou por índices/números (quando for um *array*).

Além disso, tem algumas outras regras, nos objetos os atributos devem seguir de um caractere : (dois-pontos) e o valor do atributo. E os atributos devem ser separados por vírgulas.

Já os *arrays* só podem ser de um determinado tipo de dados (veja a próxima sessão), não pode misturar texto com número, por exemplo.

>> O que são Vetores e Matrizes (*arrays*)

Veja abaixo um exemplo de objeto e um exemplo de *array*.

```
{
  "atributo1": "valor1",
  "atributo2": 2,
  "atributo3": true
}
```

[2,4,5,6]

Outro ponto interessante é que um objeto *JSON* pode ter atributos do tipo *array* e um *array* pode ser do tipo objeto ou *array*. Confuso? Veja o exemplo abaixo para entender melhor.

```
{
  "atributoDoTipoArray" : [1,2,3,54]
}
```

```
[{
  "a":1
},{
  "b":1
}]
```

Uma observação é que tanto *array* quanto objeto podem ser vazios em *JSON*. Assim:

```
{
  }
[]
```

5.3.2 Os tipos de dados do *JSON*

Além de objeto e *array* serem considerados os tipos de dados principais. O *JSON* também tem os tipos de dados primitivos que nós já falamos aqui no { Dicas de Programação }.

Os tipos de dados básicos do *JSON* são:

- **String** - separados por aspas (duplas ou simples). Ex. "Brasil" ou 'Brasil'
- **Número** - sem aspas e pode ser inteiro ou real, quando for do tipo real deve-se usar o caractere . (ponto) para separar a parte inteira das casas decimais. Ex. 1 (inteiro) ou 23.454 (real)
- **Booleano** - tipo lógico normal, pode assumir valores *true* ou *false*.
- **Nulo** - este é o valor nulo mesmo. Ex. { "nome" : *null* }

Veja abaixo um exemplo de objeto *JSON* com todos estes tipos de dados.

```
{
  "texto" : "Brasil",
  "numero" : 23,
  "numeroReal" : 54.87,
  "booleano": true,
  "nulo": null
}
```