



AULA 1

**Primeiros passos no
desenvolvimento de um sistema
*web***



O PROFESSOR

Fernando Correia

Líder de pesquisa na ExACTa (PUC-Rio) desde 2021. Doutor em informática pela PUC-Rio, onde também recebeu o título de mestre em informática, e bacharel em engenharia de computação pela Universidade Estadual de Feira de Santana. Tem vasta experiência na exploração de dados, atuando principalmente em projetos de pesquisa aplicada nas áreas de processamento de linguagem natural, extração de informação e aprendizado de máquina. Na ExACTa, lidera projetos em cocriação com grandes empresas, como Petrobras e Americanas SA. Foi pesquisador associado da Escola de Direito da Fundação Getúlio Vargas (FGV-Direito Rio) por mais de cinco anos, onde também atuou como engenheiro líder do projeto Supremo em Números. Antes da FGV, trabalhou como desenvolvedor de software na JusBrasil, a maior plataforma de documentos jurídicos do país.

Objetivos de aprendizagem

Ao final desta aula, você irá:

- Identificar as restrições que devem ser observadas em um projeto de sistema web;
- Entender a importância da separação de responsabilidades no projeto de um sistema web;
- Utilizar o conhecimento no desenvolvimento de um *back-end* de sistema web simples.

Que história é essa?

O foco dessa nossa primeira aula será no projeto e desenvolvimento de um *back-end* para um sistema web. Antes disso, precisamos entender a arquitetura da web e vamos explorar esse ponto sob uma perspectiva histórica, discutindo as restrições que devem ser observadas no desenvolvimento de um sistema web. Entraremos em detalhes em algumas dessas restrições, como a uniformidade de interfaces e a separação de responsabilidades, ressaltando a importância delas a expansão da web e o papel que cada uma tem no desenvolvimento de sistemas web modernos.

Por essa ser uma disciplina de conteúdo básico, vamos nos aprofundar no tema *back-end* focando no desenvolvimento de sistemas web tradicionais como servidores de páginas web e APIs. Segundo esse contexto, discutiremos o uso de padrões de arquitetura em sistemas Web, explorando o *framework Model-View-Controller* (MVC).

Na última parte da aula, vamos praticar o desenvolvimento *back-end* de uma aplicação web utilizando *Python*.

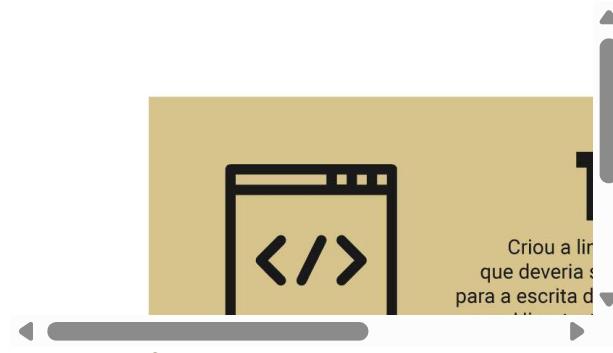
O início da web e seus desafios

É difícil separar a web da internet, ambas são tão interligadas que até esquecemos que são diferentes e se complementam. A forma mais fácil de entender é voltando no tempo.

A internet foi criada em 1969, pela Darpa, a Agência de Projetos de Pesquisa Avançada de Defesa dos Estados Unidos. Sendo assim, tinha inicialmente um propósito mais voltado para a área militar. Mais de vinte anos depois surgiu a primeira página web, em 1991, em um centro de pesquisa em Genebra, com o propósito de facilitar o compartilhamento de conhecimento via internet.

O projeto que definiu a web que conhecemos atualmente começou em 1990, na Organização Europeia para a Pesquisa Nuclear (CERN). À época, seu nome era WorldWideWeb, hoje apenas WWW (o famoso www no início da maioria das páginas web). O pai do projeto foi o britânico Tim Bernes-Lee, que não só criou a web como também

pavimentou o caminho para sua evolução. Clique na seta e veja as outras criações do Tim Bernes-Lee:



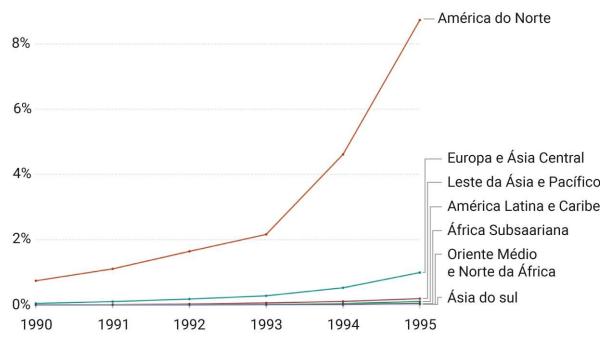
Interativo

Aqui, vemos uma caixa amarela, com um ícone preto mostrando uma janela de navegador com um símbolo de código sinal de menor do que barra sinal de maior do que. Ao lado, lê-se o texto: 1. Criou a linguagem que deveria ser utilizada para a escrita de documentos em Hipertextos, o HTML (HyperText Mark-up Language). A seguir, vemos uma caixa vermelha, com um ícone amarelo de documento com o texto 'URL' e um globo terrestre. Ao lado, lê-se: 2. Criou a sintaxe de como documentos HTMLs e outros recursos deveriam ser identificados na Web via uma URI (Uniform Resource Identifier). A seguir, vemos uma caixa azul, com um ícone amarelo com um globo terrestre e uma caixa de pesquisa com uma lupa ao seu lado. Ao lado, lê-se o texto: 3. Implementou como visualizar os recursos disponibilizados na web com o primeiro browser. A seguir, vemos uma caixa vermelha com um ícone amarelo mostrando uma janela de programação com um símbolo menor do que barra maior do que, uma engrenagem e painéis de controle. Ao lado, lê-se o texto: 4. Implementou como editar e criar novos conteúdos com o primeiro editor HTML. A seguir, vemos uma caixa cinza com um ícone vermelho mostrando um globo terrestre conectado por duas linhas a dois dispositivos: à esquerda, um monitor de computador, e à direita, um smartphone. Ao lado, lê-se o texto: 5. Criou o protocolo de transferência de hipertextos, o HTTP (HyperText Transfer Protocol). A seguir, vemos uma caixa azul, com um ícone amarelo com uma mão segurando um globo terrestre. Ao lado, lê-se um texto: 6. Criou o primeiro servidor Web.

Após a criação da *web*, o número de usuários da internet cresceu exponencialmente, saindo de 2,6 milhões em 1990 e ultrapassando 44,4 milhões em 1995 (MASSE, 2011).

A imagem a seguir ilustra esse crescimento. Essa rápida evolução foi o retrato do sucesso, mas também uma grande preocupação. O principal ponto foi a qualidade da infraestrutura da internet à época, que não oferecia suporte para esse ritmo. Confira:

Utilização da internet entre 1990 e 1995



Adaptado de University of Oxford (2020).

Para contornar esse cenário, em 1993, Roy Fielding, cofundador do Apache HTTP Server Project, propôs um caminho para a continuidade da expansão da web. Sua proposta era que a escalabilidade da web fosse governada por um conjunto de *key constraints* (restrições/limitações-chave). Segundo ele, a continuidade da expansão seria possível se cada uma dessas restrições fosse atendida de forma igual.

Essas restrições/limitações ainda são válidas e devem ser observadas e atendidas no desenvolvimento de qualquer sistema web (COOKSEY, 2014). Fielding as categorizou em seis grupos:



Vamos abordar um pouco essas restrições nos tópicos a seguir.

Cliente-servidor

Essa restrição é relativa à separação de responsabilidades na web entre aplicações cliente e servidor. Elas têm responsabilidades diferentes, mas complementares. Além disso, podem ser implementadas e publicadas separadamente, independentemente das tecnologias utilizadas em seu desenvolvimento, desde que existam interfaces uniformes de comunicação entre elas. Portanto,

não importam a linguagem, a tecnologia ou o comportamento interno, desde que haja uma forma conhecida de comunicação. Descubra mais a seguir:

Cenário de interação entre três aplicações: browser que acessa a web, servidor que mantém a aplicação e servidor de dados. Este é um cenário comum e ao longo do curso iremos explorar esses e outros cenários. Clique nos

①



Interativo

Cenário de interação entre três aplicações: browser que acessa a web, servidor que mantém a aplicação e servidor de dados. Este é um cenário comum e ao longo do curso iremos explorar esses e outros cenários. No contexto desta aula, isto também ilustra como pode se dar o relacionamento cliente-barra-servidor. O servidor da aplicação serve ao cliente (browser) um conteúdo web, ao mesmo tempo em que assume o papel de cliente para o servidor de dados. Ou seja, nesse cenário, o servidor da aplicação consome informações da base de dados e as retorna para o usuário final via browser (cliente da aplicação). Aqui, vê-se uma imagem que mostra o cliente (representado por um desenho de uma aba de navegador com uma imagem do globo terrestre aberta) conectado ao servidor, que contém camada 1 (servidor da aplicação), cliente, e a camada 2, servidor de dados, servidor, que é conectado ao banco de dados (representado pelo desenho de três anilhas empilhadas).

Uniformidade de interfaces

Como ilustrado na restrição de arquitetura cliente/servidor, a comunicação entre aplicações depende da uniformidade de suas interfaces, ou seja, a forma de comunicação deve ser conhecida entre as partes. Esse ponto é tão crucial que vários padrões foram e ainda são definidos para auxiliar essa uniformização. Fielding destaca quatro pontos que devem ser observados:

Clique nos cards para saber mais.



Interativo

Aqui, vê-se quatro caixas numeradas de 1 a 4. A 1 e a 3 são amarelas, enquanto a 2 e a 4 são vermelhas. Ao clicar em cada uma delas, vê-se o conteúdo a seguir:

- 1. Identificação de recursos. Todo e qualquer conteúdo disponibilizado na web é definido como um recurso (páginas HTML e arquivos de imagem ou vídeo são exemplos de recursos), devendo ter um identificador único. O URL, um padrão de identificação única de recursos, define a sintaxe que deve ser obedecida para a criação de um novo URI. Veja dois exemplos de URLs: (a) ftp dois pontos barra barra ftp ponto is ponto co ponto za barra rfc barra rfc1808 ponto txt (b) http dois pontos dabliu dabliu dabliu ponto ietf ponto org barra rfc barra rfc 2396 ponto txt A primeira (a) identifica um arquivo de texto e o protocolo FTP como o que deve ser utilizado para a transferência de conteúdo. A segunda (b) também é relativa a um arquivo de texto, mas o protocolo a ser utilizado é o HTTP. Ambas URLs identificam de forma única Web os acessos aos recursos 'rfc1808 ponto txt' e 'rfc2396 ponto txt', respectivamente.
- 2. Manipulação de recursos via representações. O objetivo é manipular sempre a representação do recurso, nunca o recurso em si. Um exemplo simples é a visualização de vídeos em aplicativos de stream: um mesmo vídeo (recurso) poderá ser visualizado via diferentes browsers e dispositivos, com o identificador do vídeo (seu URL) sendo sempre o mesmo.
- 3. Uso de mensagens autodescritivas. A informação de estado sobre a requisição de um recurso deve ser sempre dada na resposta ao cliente. No protocolo HTTP, mensagens sobre o estado da requisição são fornecidas no cabeçalho da resposta. O HTTP define uma coleção de códigos relacionados a cinco categorias de mensagens: informacional, sucesso, redirecionamento, erro do cliente e erro de servidor. Esses códigos e mensagens estão especificados no RFC 2616, o mesmo documento que define o protocolo HTTP.
- 4. Hipermídia como motor de estados de aplicação (Hateoas). A representação do estado atual de um recurso inclui os links aos recursos relacionados. O objetivo é que o cliente não necessite conhecer todos os recursos disponíveis de uma aplicação, apenas um URL inicial, e, a partir dos links fornecidos na página inicial, acesse todos os outros recursos disponíveis de forma circular, podendo se guiar pelas requisições realizadas. Um exemplo seria a busca por um produto navegando pelos departamentos de uma loja on-line, partindo da página inicial. O URL final seria algo do tipo: http dois pontos barra barra loja ponto com ponto br barra departamento underline a barra produto underline a Neste URL, 'loja ponto com ponto br' está destacado em preto, 'departamento underline a' está destacado em azul e 'produto underline a' está destacado em cinza. Nesse tipo de navegação, não é necessário que o cliente conheça de antemão todos os URLs possíveis da loja. É importante que a aplicação guie o cliente por meio dos links, auxiliando na busca por um recurso desejado.

Sistema em camadas

Essa restrição de arquitetura da web existe para permitir que serviços intermediários possam ser criados para atuar entre o cliente e o servidor sem impactar a experiência do cliente. Um exemplo prático e simples é o *proxy*, que age como um intermediário entre cliente e servidor. Uma vez configurado o *proxy*, o serviço se torna transparente e o cliente acessa o recurso de interesse sem que seja nítida sua presença.

A restrição de arquitetura em camadas também pode servir para outros propósitos. Por exemplo, na imagem a seguir, temos um sistema em três camadas: o cliente (representado pelo *browser*) está na camada 0; o servidor da aplicação está na camada seguinte; e o acesso aos dados só é possível na terceira camada. Nessa arquitetura, o cliente não tem acesso direto aos dados, pois não estão em camadas vizinhas. Ainda assim, é transparente para o cliente a existência da terceira camada. Veja a seguir:

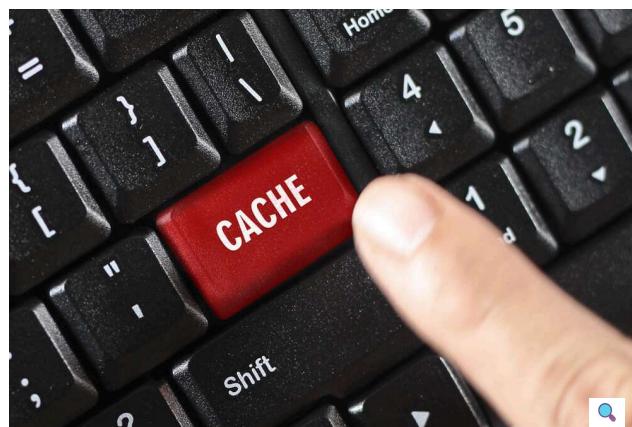
Um sistema em três camadas



Esse tipo de arquitetura faz com que a camada 2 possa servir como camada de segurança, legitimando quem pode ter acesso aos dados e até que ponto eles podem ser acessíveis (por exemplo, limitando o acesso de acordo com o contexto fornecido).

Cache

Esse é um dos pontos mais importantes da arquitetura web. O objetivo dessa restrição é exigir da aplicação servidor que declare se o dado enviado em uma resposta a uma requisição é “cacheável”, ou seja, se pode ser armazenado em memória para evitar que outra requisição que o peça seja feita em curto prazo de tempo.



O cache de uma resposta pode ajudar na “percepção de latência” do cliente, aumentando a disponibilidade de um recurso e reduzindo a carga do servidor da aplicação.

O cliente que precisa acessar um mesmo recurso mais de uma vez vai perceber rapidez na resposta, quando, na verdade, o recurso foi guardado no *cache* do browser ou em qualquer outro servidor/provedor de internet que possa existir no caminho até o servidor. Do outro lado, foram várias requisições do cliente, mas para o servidor da aplicação foi apenas uma resposta. Isso representa uma economia de recurso e maior disponibilidade para atender a outras requisições e clientes.

Stateless (ausência de estado)

Essa restrição de arquitetura determina que um servidor web não necessita memorizar os estados da aplicação cliente. Como resultado, o cliente deve incluir toda informação de contexto que possa ser considerada relevante na requisição, para qualquer iteração com o servidor. Livre dessa preocupação, um servidor pode atender a diferentes clientes de forma ágil.



Quanto mais complexo o contexto e maior a informação enviada, menor a capacidade de atendimento do servidor. Balancear esses dois fatores é importante para contribuir com a escalabilidade da aplicação servida.

Code-on-demand

Essa restrição de arquitetura da web permite que servidores possam enviar pequenos trechos executáveis de código, como *scripts* e *plugins*, para clientes. O objetivo é permitir que a aplicação servidora possa oferecer um meio para a aplicação cliente interpretar um dado enviado. Com isso, o código enviado habilitaria o cliente a interpretar o dado enviado pelo servidor.



Essa restrição é a única que pode ser considerada opcional entre todas as apresentadas. Ela é muito comum quando se utilizam tecnologias como Java Applets e JavaScripts.

O protocolo HTTP

Como apresentado no tópico anterior, os trabalhos de Fielding e Berners-Lee pavimentaram o caminho para o crescimento da web, que continua em plena expansão. Neste curso, vamos nos restrinir ao desenvolvimento de sistemas que atuam no ambiente web e utilizam o protocolo HTTP para comunicação.

```
36 </style>
37 </head>
38 <body>
39 <div class="main">
40 <div class="header">
41 <div class="block_header">
42 <div class="logo"><a href="index.html">WPS</a>
43 <div class="menu">
44 <ul>
45 <li><a href="index.html">Welcome</a>
46 <li><a href="services.html">About us</a>
47 <li><a href="services.html">Services</a>
48 <li><a href="portfolio.html">Portfolio</a>
49 <li><a href="contact.html">Contact</a>
50 </ul>
51 </div>
<div class="clr"></div>
```

O protocolo HTTP é um dos mais adotados no desenvolvimento de sistemas web. Um de seus benefícios é a facilidade de aprendizado. Além disso, apresenta uma série de funcionalidades nativamente aderentes às restrições de arquiteturas mencionadas anteriormente – vale lembrar que as pessoas responsáveis pela versão atual desse protocolo também foram as que criaram a web. Assim, vamos explorá-las ao longo do curso.

Mantendo o foco

A comunicação em HTTP reside no conceito de **ciclo de requisição e resposta**: o cliente envia uma requisição para o servidor com a expectativa de receber uma resposta (BIEHL, 2011).

Para fazer uma requisição em HTTP a aplicação cliente precisa informar:

Clique nos cards para saber mais.



Interativo

Aqui, vê-se quatro caixas numeradas de 1 a 4. As caixas 1 e 3 são amarelas e as caixas 2 e 4 são azuis. Clicando em cada uma delas, vê-se o seguinte conteúdo: 1. URL (localizador-padrão de recursos, ou uniform resource locator). É um URL com o endereço de um recurso na internet, que pode fazer referência tanto a uma página HTML como a um áudio, vídeo ou imagem. 2. Método. Informa ao servidor o tipo de ação que o cliente quer que este tome. São cinco possíveis métodos, sendo os mais comuns: GET, para solicitar o retorno de um recurso; POST, para solicitar a criação de um recurso; PUT, para solicitar a edição ou atualização de um recurso; DELETE, para solicitar que seja deletado um recurso. 3. Cabeçalho. Provê uma lista de metainformações sobre a requisição. Um exemplo de metadado informado no cabeçalho é o user traço agent, que ajuda o servidor a identificar o tipo de browser ou dispositivo que o cliente está usando para fazer a requisição. Com essa informação, o servidor pode decidir como será construída a resposta para oferecer ao cliente a melhor experiência possível. Por exemplo, ao acessar um site de stream de vídeo, o user traço agent vai ajudar o servidor a determinar o melhor formato de vídeo a ser apresentado na resposta. 4. Corpo da requisição. Contém o dado que o cliente quer enviar para o servidor. Um detalhe importante sobre esse campo é que o cliente tem total controle sobre o conteúdo a ser enviado, diferentemente dos outros campos, que têm uma sintaxe bem definida.

Depois que o servidor receber a requisição, irá tentar fazer o que foi solicitado e retornar uma resposta. Essa resposta terá uma estrutura similar à da requisição, mas, em vez do URL e do método, informará o *status code*, que, como já mencionado, é um valor numérico que representa uma mensagem autodescritiva que informa o estado final da execução do que foi solicitado. Veja um resumo dos diferentes códigos e seus significados:

Faixa de códigos	Classe	Exemplo
100-199	Informacional	102 Processing. O servidor recebeu a requisição e está processando.
200-299	Sucesso	200 OK. A resposta foi retornada com sucesso.
300-399	Redirecionamento	302 Found. O URL solicitado foi alterado temporariamente.
400-499	Erro na resposta	400 Bad Request. O servidor não pode retornar uma resposta válida, geralmente devido a um problema de autenticação.
500-599	Erro de servidor	500 Internal Server Error. O servidor encontrou um problema e não sabe como proceder.



O cabeçalho também é alterado na resposta do servidor, passando informações relevantes para a construção da representação da resposta. Por exemplo, o dado *Content-type* informa o tipo de dado retornado para indicar ao cliente como interpretá-lo. A imagem a

seguir apresenta um resumo comparativo da estrutura de requisição e resposta:



Padrões de arquitetura em sistemas web

Um sistema web é um *software* e, sendo assim, sua arquitetura deve ser bem planejada para facilitar seu desenvolvimento, manutenção e evolução. A importância de uma boa arquitetura é fácil de justificar, pois define a base para o desenvolvimento da aplicação, influenciando diretamente diferentes aspectos do sistema, como mostrado por Biehl (2011). Veja a seguir, clicando nos títulos:

Design

Monetização

Confiabilidade

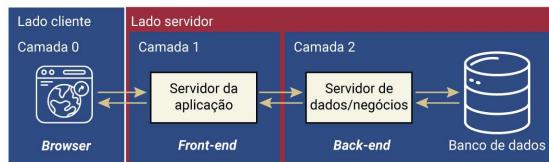
Escalabilidade

Segurança

O projeto da arquitetura deve ser guiado com base nos objetivos esperados para a aplicação. Como a importância de cada um desses aspectos pode variar com o projeto, a arquitetura do sistema deve ser aderente a seus objetivos. No entanto, vale ressaltar que ela também pode evoluir. Sendo assim, a evolução da aplicação é mais um aspecto a ser considerado.

Com isso, não há um padrão de arquitetura, mas vários, e a escolha vai depender das necessidades e objetivos do sistema. Neste curso, por ser um módulo básico, vamos focar uma estratégia simples de arquitetura web de apenas três camadas. Vamos explorar a separação de responsabilidade e o respeito às restrições de arquiteturas mencionadas anteriormente. Veja como será destacado na arquitetura deste sistema:

Arquitetura em três camadas



Nessa proposta de arquitetura, duas aplicações do lado do servidor, em duas camadas diferentes, tratarão de responsabilidades específicas: uma servirá a apresentação (*front-end*) e a outra fará a execução de lógicas de negócios e o envio de dados.

Essa simples separação de responsabilidades pode ser suficiente para a construção de um sistema web escalável. A persistência e a leitura de dados em banco geralmente têm um custo elevado de tempo, pelas consultas realizadas. Por outro lado, a construção da representação, da página final que será respondida ao *browser*, tem, geralmente, um custo bem menor de tempo e processamento. Com isso, essa separação permite que, em um cenário de alta demanda, seja possível publicar mais de uma instância do servidor de dados, para aumentar a disponibilidade geral do lado deste.

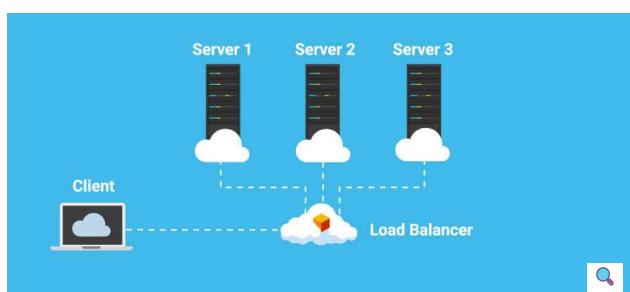
Supondo um cenário de alta demanda

Clique nos títulos e entenda melhor:

[Servidor](#)

[Dados](#)

[Resultado](#)



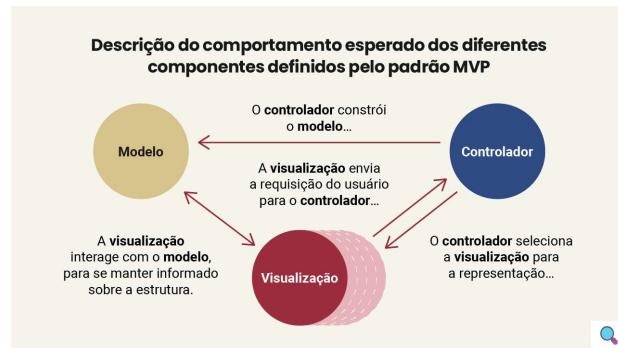
Para mais de uma instância de um mesmo servidor publicado na web, é necessário um componente para o balanceamento de cargas, o **load balancer**. Graças às restrições de ausência de estados e de sistemas em camada, o *load balancer* é transparente para o cliente (assim como um proxy) e age como um gerenciador de fluxos. Ele recebe a requisição do cliente e a encaminha para uma instância do servidor.

que está disponível. Depois de processada a requisição, a resposta é enviada para o *load balancer* e encaminhada para o cliente.

Padrão MVC

Uma preocupação recorrente no projeto de arquitetura de um sistema web é a separação de responsabilidades e a uniformização das interfaces. Nesse sentido, o modelo visualização e controlador (MVC – *model-view-controller*) é um padrão de projeto de arquitetura de software muito aderente ao desenvolvimento web.

Em termos de flexibilidade, não importa de onde o conteúdo vem ou como é enviado, mas, sim, o **modelo** do dado, como ele é representado. Sabendo como é a estrutura do dado, o conteúdo pode ser transformado em uma representação (ou **visualização**), de acordo com o mecanismo de controle (o **controlador**) encarregado de selecionar o dado, definir o modelo e escolher a melhor visualização (SHKLAR; ROSEN, 2009). Confira nesta imagem:



Assista, a seguir, a um vídeo que trata do assunto:

Separação de responsabilidades no desenvolvimento web

O padrão MVC é comum no desenvolvimento de softwares, mas como pode ser utilizado em sistemas web? Essa discussão vai longe e, neste vídeo, veremos um caminho possível para a aplicação do MVC no desenvolvimento de um sistema API.



Na seção Técnica Aplicada, vamos explorar um pequeno exemplo de um sistema web desenvolvido em Python.

Estudo complementar

Para ampliar seus conhecimentos, leia o capítulo “Construção do modelo MVC de um cenário real”, você vai aprender a desenvolver o modelo MVC no respectivo cenário, em conjunto com um banco de dados.

Para ler este livro, busque o *link* no card **leituras**, dentro da disciplina no AVA.

Livro

Programação back-end II

Ao rever a aula, podemos ter em mente que a versão atual do protocolo HTTP é definida e aderente às restrições de arquitetura da web. Além disso, a escalabilidade é um fator-chave para qualquer API.

Chegou o momento de olhar de maneira crítica os primeiros passos no desenvolvimento de um sistema web e colocar em prática todo o seu conhecimento. Veja a seguir:

Reflita!

Em um mundo cada vez mais dirigido por decisões baseadas em dados, é crescente a popularização de APIs. Um bom exemplo é a API do Twitter, que permite o acesso a um gigantesco volume de mensagens em tempo real.



Você acha que a popularização das APIs pode tornar caótica a web atual?

Seguindo nesse contexto, é fácil compartilhar notícias nas redes sociais. Basta estar em um portal de notícias, clicar em Compartilhar (provavelmente, haverá um *link* bem visível para isso) e escolher a rede social. Geralmente, esse ato é facilitado por uma API fornecida pela rede social, que facilita a comunicação quando ambas as plataformas (o portal de notícias e a rede social) têm o mesmo usuário em comum.

A partir disso, reflita quais são os desafios no desenvolvimento desse tipo de API.

Referências

Clique aqui para acessar as referências e créditos desta aula.