**Aula XXI - Desenvolvimento Web III**

**Arquitetura de Software**

**Planejando e Projetando uma API**

**Design & Versionamento**

**Padrão MVC**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***Fontes:***

[*https://distancia.qi.edu.br/mod/book/view.php?id=31580*](https://distancia.qi.edu.br/mod/book/view.php?id=31580)

[*https://medium.com/@oieduardorabelo/arquitetura-de-software-a-diferen%C3%A7a-entre-arquitetura-e-design-964ab38ea61a*](https://medium.com/@oieduardorabelo/arquitetura-de-software-a-diferen%C3%A7a-entre-arquitetura-e-design-964ab38ea61a)

[*https://imasters.com.br/desenvolvimento/arquitetura-e-desenvolvimento-de-software-parte-01-introducao*](https://imasters.com.br/desenvolvimento/arquitetura-e-desenvolvimento-de-software-parte-01-introducao)

[*https://code.tutsplus.com/series/design-patterns-in-php--cms-747*](https://code.tutsplus.com/series/design-patterns-in-php--cms-747)

[*https://phptherightway.com/pages/Design-Patterns.html*](https://phptherightway.com/pages/Design-Patterns.html)

[*https://becode.com.br/etapas-de-um-projeto-de-software/*](https://becode.com.br/etapas-de-um-projeto-de-software/)

[*https://www.ibm.com/developerworks/br/opensource/library/os-cloud-saas/index.html*](https://www.ibm.com/developerworks/br/opensource/library/os-cloud-saas/index.html)

[*https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f*](https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f)

[*https://www.devmedia.com.br/criando-apis-de-desenvolvimento-de-software/30445*](https://www.devmedia.com.br/criando-apis-de-desenvolvimento-de-software/30445)

[*https://www.devmedia.com.br/qualidade-no-codigo-java-com-boas-praticas-e-clean-code/39472*](https://www.devmedia.com.br/qualidade-no-codigo-java-com-boas-praticas-e-clean-code/39472)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Arquitetura de Software**

A arquitetura de software trata um sistema como algo a ser construído, preocupando-se com sua organização e com a estrutura geral.

**Processo de Arquitetura de Software**

Os processos da fase de planejamento e projeto de software envolvem:

* Elaboração do modelo de negócio – análise de custo, tempo de desenvolvimento, restrições de mercado, interfaces com outros sistemas, etc;
* **Entendimento dos requisitos: levantamento de requisitos e modelo do domínio;**
* **Criação ou seleção de uma arquitetura: identificação dos componentes e suas interações, das dependências de construção e tecnologias que apoiam a implementação;**
* **Representação da arquitetura e divulgação: para permitir aos desenvolvedores e testadores o entendimento da arquitetura;**
* Implementação da arquitetura: guiar o desenvolvimento seguindo seus protocolos e estruturas;
* Análise e avaliação: verificar a adequação da arquitetura, registrando impactos, riscos e dificuldades, o que servirá para evolução da arquitetura.

**As características da arquitetura de software**

**As características do software descrevem os requisitos e as expectativas de um software nos níveis operacional e técnico.** Assim, quando um líder de produto diz que está competindo em um mercado de rápida mudança e deve adaptar seu modelo de negócios rapidamente. **O software deve ser “extensível, flexível, modular e de manutenção adequada” se uma empresa lidar com solicitações urgentes que precisam ser concluídas com êxito em questão de tempo.** Como arquiteto de software, observe que o desempenho e a baixa tolerância a falhas, escalabilidade e confiabilidade são suas principais características. Agora, depois de definir as características anteriores, o proprietário da empresa informa que eles têm um orçamento limitado para esse projeto, outra característica aparece aqui, que é **“a viabilidade”**.

**Padrões de arquitetura de software**

A maioria das pessoas provavelmente já ouviu falar do termo **“MicroServices”** antes. O MicroServices é um dos muitos outros padrões de arquitetura de software, como padrão em camadas, padrão orientado a eventos, padrão serverless e muitos mais. Alguns deles serão discutidos mais adiante neste artigo. O padrão de micros-serviços ganhou reputação após ser adotado pela Amazon e Netflix e mostrar seu grande impacto.

**Arquitetura serverlets**

Essa arquitetura refere-se à solução de aplicativo que depende de serviços de terceiros para gerenciar a complexidade dos servidores e o gerenciamento de back-end. Arquitetura serverless é dividida em duas categorias principais. O primeiro é “Back-end como serviço (BaaS)” e o segundo é “Funções como serviço (FaaS)”. A arquitetura sem servidor ajudará você a economizar muito tempo cuidando e corrigindo erros nas tarefas regulares de implantação e servidores. O provedor mais famoso de API sem servidor é o Amazon AWS “Lambda”.

A infraestrutura para desenvolvimento Web em Java é baseada em Servlets. Hoje chamamos de infraestrutura, mas, nos primórdios, elas eram usadas diretamente para desenvolver aplicações. **Uma Servlet é uma classe Java** que fica no servidor e é invocada com base em algum padrão de URL. O nome **“servlet”** vem do inglês e dá uma ideia de servidor pequeno cujo objetivo basicamente é receber requisições HTTP, processá-las e responder ao cliente, essa resposta pode ser um HTML, uma imagem etc.

**Em Java, procuramos sempre trabalhar orientado a objeto, portanto essa nosso servlet será também um objeto de uma classe Java.**

|  |
| --- |
| **Serviços baseados no protocolo HTTP(s) (Hypertext Transfer Protocol - Security) - URL ou URI (http://algumnome/nome)** |

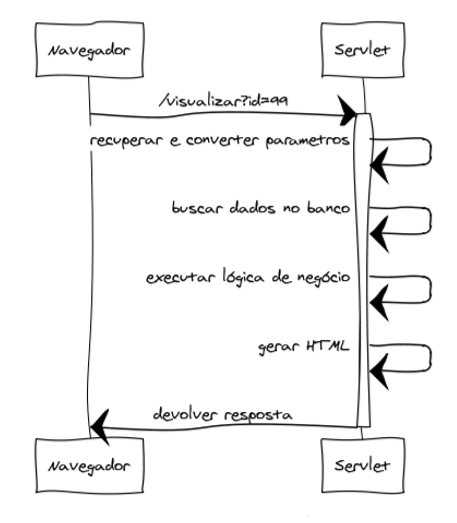
**O funcionamento se dá da seguinte forma:**

* Cliente (navegador) faz uma requisição HTTP ao servidor.
* O servlet responsável trata a requisição e responde ao cliente de acordo.
* O cliente recebe os dados e exibe.

Para esse artigo abordaremos os comportamentos de servlet que foram definidos na classe HttpServlet do pacote javax.servlet, que é usada em quase em quase 99% dos casos.

Para criarmos uma classe **servlet ela terá que ser subclasse de HttpServlet** e sobrescrever o **método service(),** que é responsável por receber as requisições e retornar suas respostas. O modificador **protected torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.**

**Fluxo de execução que mostra onde tudo fica no Servelet**

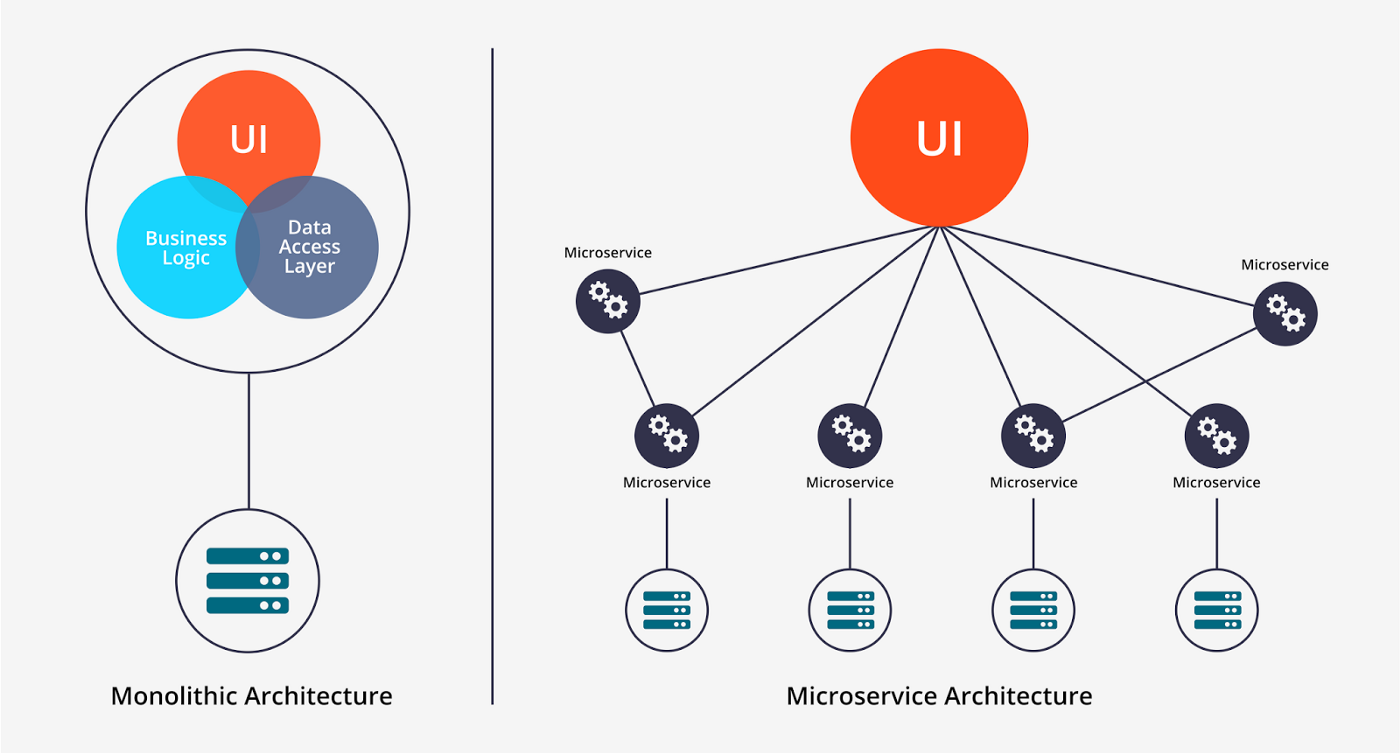
**Arquitetura orientada a eventos**

Essa arquitetura depende de produtores e consumidores de eventos. **A idéia principal é desacoplar as partes do sistema e cada parte será acionada quando um evento interessante de outra parte for acionado.** Vamos simplificar: **Suponha que você crie um sistema de loja online e ele tenha duas partes. Um módulo de compra e um módulo de fornecedor. Se um cliente fizer uma compra, o módulo de compra geraria um evento “orderPending”. Como o módulo do fornecedor é interessado no evento “orderPending”, ele estará ouvindo, caso um evento desse seja acionado.** Quando o módulo do fornecedor obtiver esse evento, ele executará algumas tarefas ou talvez acionará outro evento para solicitar mais do produto a um determinado fornecedor.

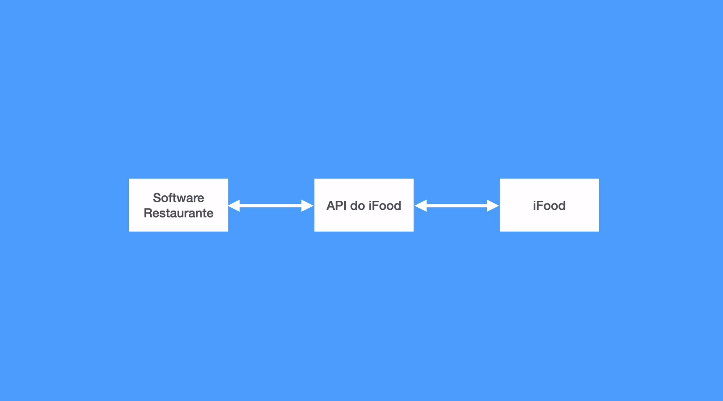
Lembre-se de que o produtor do evento não sabe qual consumidor do evento está ouvindo qual evento. Além disso, outros consumidores não sabem quais deles ouvem quais eventos. Portanto, a idéia principal é dissociar as partes do sistema.

**Arquitetura de micros-serviços**

**A arquitetura de micros-serviços se tornou a arquitetura mais popular nos últimos anos.** Depende do desenvolvimento de serviços modulares pequenos e independentes, em que cada serviço resolve um problema específico ou executa uma tarefa exclusiva e esses módulos se comunicam por meio de uma API bem definida para atender à meta de negócios.



**Exemplo:** Se você tem um restaurante e desenvolve um sistema para poder vender os itens do seu cardápio, poderá usar os micros - serviços do iFood, por exemplo. **Graças a uma API, vários restaurantes podem automatizar a venda de seus produtos.**



**O que é e Projetando uma API**

As API’s são a base para qualquer negócio informatizado, portanto são a base para qualquer empresa, por isso precisamos abordar os conceitos básicos sobre o **padrão Rest, sobre o protocolo HTTP e uma série de boas práticas visando a construção de uma API Rest de alto nível.**

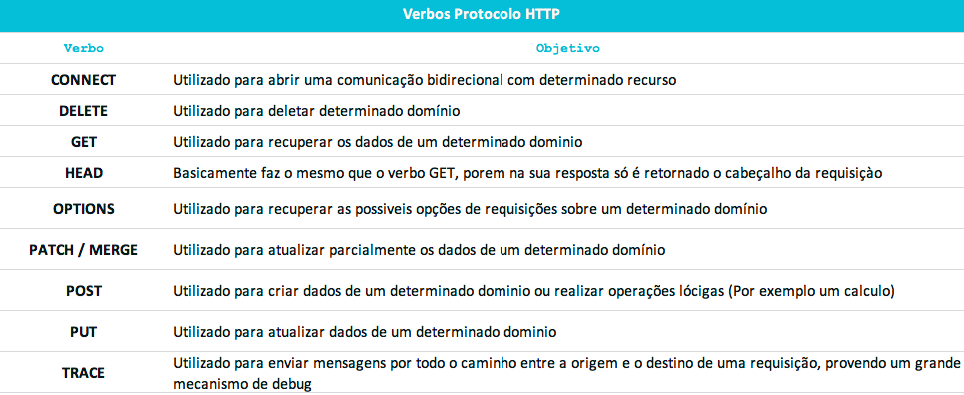
**Rest e Restful**

**Representational State Transfer** (Transferência de Estado Representacional) ou Rest, representa uma série de princípios definidos pela **WWW**, visando a padronização de rotas, requisições e comunicações sem estado, o próprio protocolo **HTTP**, foi baseado nestas regras e se propõe a resolver todos estes problemas. **Restful é o termo atribuído a uma API que possui a inteligência de aplicar o padrão Rest.**

**Protocolo HTTP**

O protocolo HTTP é a base de tudo, as requisições chegam e saem através dele, os padrões são baseados em sua estrutura, e seus códigos de resposta, por esse motivo, **todas as API's devem respeitá-lo e aplicar suas melhores práticas, portanto segue abaixo algumas delas:**

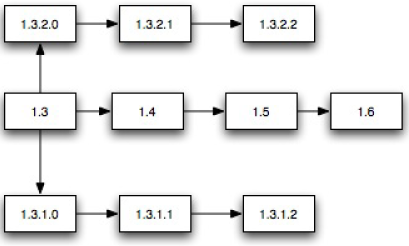
**Métodos do protocolo HTTP**

*Fonte da imagem:* [*https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f*](https://medium.com/@wssilva.willian/design-de-api-rest-9807a5b16c9f)

A maioria das Api's utilizam apenas os métodos **GET, POST, PUT e DELETE**, porém isso não é uma regra, se fizer sentido para sua aplicação, utilize-as.

**Versionamento**

O sistema de numeração utilizado para sistemas de software é baseado em números naturais com um ponto os separando. Este ponto é necessário para acomodar “não-linearidades” no desenvolvimento de software. Isso ocorre porque não existe uma direção única de desenvolvimento de um software, mas sim muitos ramos representando correção de bugs, correção de um bug que gerou outro bug, e assim por diante. Dessa forma, uma versão 1.1.1 significa que ela contém menos funcionalidades que uma versão 2.0 que foi lançada antes da versão 1.1.1.



**O visual e a documentação do Projeto de uma API**

Os diagramas de arquitetura e microarquitetura de software são úteis para estabelecer e comunicar uma base para um cliente em potencial e para oferecer às equipes de desenvolvimento uma estrutura de tópicos para seguirem ao organizar o código. Esses diagramas também estabelecem um conjunto de diretrizes e expectativas a partir de um ponto de vista estrutural.

Podemos trabalhar partindo de um fluxograma, um diagrama de classes ou mesmo o efetivo diagrama MVC que exibe os três tópicos separados do desenvolvimento de uma API.

|  |  |  |
| --- | --- | --- |
| **Diagrama de Arquitetura MVC** | **Diagrama de Arquitetura MVC** | **Diagrama de Relações** |
|  |  |  |
| **Diagrama de Classe** | **Fluxograma de uma API** | |
|  |  | |

**O padrão MVC**

O modelo de três camadas físicas divide um aplicativo de modo que a lógica de negócio resida no meio das três camadas físicas. Isto é chamado de camada física intermediária ou camada física de negócios. A maior parte do código escrito reside na camada de apresentação e de negócio.

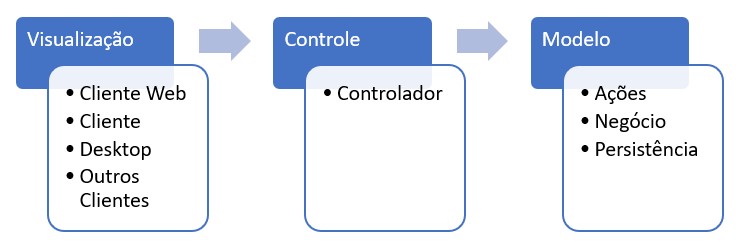
**A arquitetura MVC - (Modelo Visualização Controle) fornece uma maneira de dividir a funcionalidade envolvida na manutenção e apresentação dos dados de uma aplicação. A arquitetura MVC não é nova e foi originalmente desenvolvida para mapear as tarefas tradicionais de entrada , processamento e saída para o modelo de interação com o usuário. Usando o padrão MVC fica fácil mapear esses conceitos no domínio de aplicações Web multicamadas.**

Na arquitetura MVC o modelo representa os dados da aplicação e as regras do negócio que governam o acesso e a modificação dos dados. O modelo mantém o estado persistente do negócio e fornece ao controlador a capacidade de acessar as funcionalidades da aplicação encapsuladas pelo próprio modelo.

Um componente de visualização renderiza o conteúdo de uma parte particular do modelo e encaminha para o controlador as ações do usuário; acessa também os dados do modelo via controlador e define como esses dados devem ser apresentados.

Um controlador define o comportamento da aplicação , é ele que interpreta as ações do usuário e as mapeia para chamadas do modelo. Em um cliente de aplicações Web essas ações do usuário poderiam ser cliques de botões ou seleções de menus. As ações realizadas pelo modelo incluem ativar processos de negócio ou alterar o estado do modelo. Com base na ação do usuário e no resultado do processamento do modelo , o controlador seleciona uma visualização a ser exibida como parte da resposta a solicitação do usuário. Há normalmente um controlador para cada conjunto de funcionalidades relacionadas.

**A arquitetura de 3 camadas que está representada abaixo é uma implementação do modelo MVC. O modelo MVC está preocupado em separar a informação de sua apresentação.**



**Vantagens do modelo MVC :**

* Como o modelo MVC gerencia múltiplos visualizadores usando o mesmo modelo é fácil manter , testar e atualizar sistemas múltiplos
* É muito simples incluir novos clientes apenas incluindo seus visualizadores e controles
* Torna a aplicação escalável
* É possível ter desenvolvimento em paralelo para o modelo , visualizador e controle pois são independentes.

**Camadas do Modelo MVC**

**A Camada View (Visualização) do MVC:**

**É a camada que exibe uma representação dos dados.**

É camada de interface com usuário (view).

Também conhecida como cliente-side.

Faz a exibição dos dados, utilizando-se de #HTML e/ou XML.

É responsável por usar as informações modeladas para produzir interfaces de apresentação conforme a necessidade.

**A Camada Model (Modelo) do MVC:**

É a camada que contém a estrutura de dado atrás de uma parte específica da aplicação

Usualmente portada em JSON.

**Responsável pela leitura manipulação e validação de dados, e também de suas validações.**

**Responsável por tratar as regras de negócio.**

**Contém a lógica do negócio**

Obtém os dados e os traduz em informações relevantes para serem exibidas pela View.

**Notifica a view e controller associados quando há uma mudança em seu estado.**

**A Camada Controller (Controle) do MVC:**

**É a camada de controle.**

**Exerce o controle de qual model deverá ser aplicado e qual view será mostrado ao usuário.**

**Podemos dizer que esta camada faz uma gerência das outras duas camadas.**

O controller manipula e roteia as requisições dos usuários.

Interpreta as requisições submetidas pelo usuário e traduz em comandos que são enviados para o (Model) e/ou para a View) .

**Valida as requisições dos usuários de acordo com as regras de autenticação e autorização.**

