DAW

UNIVERSIDADE FEDERAL DE JATAÍ
Unidade Acadêmica de Ciências Exatas
Curso de Ciências da Computação

UFJ
UNIVERSIDADE FEDERAL DE JATAI

Conceitos – Módulo I – Desenvolvimento de Aplicações para WEB – 2020/2 (2021)

1 – API – Aplicações WEB

Application Programming Interface – Interface de Programação de uma Aplicação

É um componente de software que possui um conjunto de funções que faz a intermediação do acesso a algum tipo de sistema. É um conceito genérico e abrangente, pois muitos componentes de software são considerados APIs. Existe uma confusão recorrente de que Web Services é a mesma "coisa" que API. Na verdade, Web Services são APIs, porém nem todas APIs são Web Services.

E, quando se usa o termo Desenvolvimento de Aplicações WEB, é necessário separar os elementos desta frase. Primeiro, "desenvolvimento" é auto explicável, ou seja, construção, implementação e termos similares explicam algo relacionado a criar... Já o termo aplicações, para o contexto deste curso, será um conjunto maior que contém APIs e outras funcionalidades, ou seja, teremos APIs e consumidores destas APIs. E, por último o termo WEB, refere-se ao meio que será usado para disponibilizar estas aplicações, seja a API em si, seja seus consumidores.

2 - REST

Representational State Transfer – Estado Representacional de Transferência

É um modelo arquitetural! O conceito de REST surge a partir de uma tese de doutorado de Roy Fielding que apresenta um conjunto de regras para que houvesse conformidade de uma API com um conceito pré-estabelecido em termos de qualidade, funcionalidade e disponibilidade. A separação entre o provedor e o consumidor, a escalabilidade, independência de linguagem, a necessidade de integração, simplicidade e padronização são algumas definições que REST se utiliza. Do ponto de vista forma, REST apresenta regras (*constraints*) bem claras: **cliente-servidor; stateless; cache; interface uniforme; código sob demanda.**

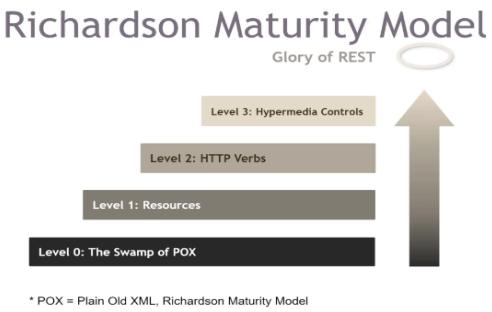
Vantagens dos *Web Services RestFul*:

- Desenvolvimento fácil e rápido.
- REST é um padrão arquitetural leve, portanto em escolhas por limitação de banda opte por Web *Services* REST.
- Rapidez nos processamentos de dados de requisições e respostas.

Segundo Roy Fielding sua API só pode ser chamada de REST API quando for HATEOAS, ou seja, quando estiver no terceiro nível de maturidade.

"O que é necessário para tornar o estilo de arquitetura REST claro sobre a noção de que o hipertexto é uma restrição? Em outras palavras, se o mecanismo do estado do aplicativo (e, portanto, a API) não estiver sendo controlada pelo hipertexto, ela não poderá ser RESTful e não poderá ser uma API REST." (Roy Fielding).

Uma forma de entender estes conceitos sobre REST, REST API, Restful é conceber que REST é uma abstração, uma especificação e que Restful é a implementação desta abstração seguindo algumas regras, que foram definidas como Modelo de Maturidade de Richardson:



Um artigo que apresenta com detalhes os conceitos de REST e seus níveis de maturidade pode ser acessado em: https://www.infoq.com/br/articles/nivelando-sua-rest-api/.

2.1 Níveis de maturidade

2.1.1 Nível 0 de Maturidade - HTTP

Nesse nível tende-se a usar somente o POST como verbo e utilizar apenas uma URI, combinando com uma variedade de comandos próprios.

Basicamente, o uso simples do protocolo HTTP apenas como comunicação com o uso de uma única URI já representa um nível 0 de maturidade.

Uma aplicação que apresenta uma interface com um botão apenas para cadastrar uma informação do usuário é exemplo neste nível.

2.1.2 Nível 1 de Maturidade - Recursos

Nesse nível, a API faz uso de URIs bem estabelecidas e os respectivos mapeamentos para cada recurso existente na aplicação. Recursos como produtos, alunos, clientes, obras, proprietários são exemplos destes mapeamentos e as URIs podem ser /produtos/v1, /clientes/nome/???? e outros.

- Nome dos recursos no plural

Manter o nome do recurso no plural é uma boa prática. Isso se justifica se pensarmos em recursos como pensamos em diretórios, e assim podemos dizer que o diretório clientes tem um arquivo com uma estrutura composta por id, nome e sexo e o identificador.

- Recurso raiz

Um “proprietário”, assim como uma “obra”, é um recurso raiz. Um recurso raiz existe independente de outro recurso. Ele aparecerá após o domínio da aplicação. HTTP://domínio/proprietarios ou HTTP://domínio/obras.

- Sub-recursos

Um sub recurso existirá após a criação do seu antecessor. Pedidos não existe sem um cliente associado a ele, ou seja, o recurso pedidos vive em função de um cliente e por esse motivo pedidos deve estar abaixo de cliente para evidenciar esse fato. Ex.: proprietarios/cpf.

2.1.3 Nível 2 de Maturidade – Verbos HTPP

Esse nível apresenta um conjunto de verbos que representam operações possíveis sobre um recurso, no primeiro nível define-se os substantivos e nesse nível aplica-se os verbos. Os principais verbos HTTP são: GET, PUT, POST, DELETE, HEAD, PATCH, OPTIONS.

```
POST    /obras    (criar)
GET     /obras/20 (buscar)
PUT     /obras/20 (atualizar)
PATCH   /obras/20 (atualizar parcialmente)
DELETE  /obras/20 (remover)
HEAD    /obras/20 (verificar se existe)
```

O retorno destas operações com estes verbos também são importantes neste nível, por exemplo, o código 200 que representa sucesso ou "OK". Porém existem outros retornos como o "created" do POST que é o código 201, além de outros códigos como 404 (NotFound) e outros...

2.1.3 Nível 3 de Maturidade – HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) basicamente são links e aplicação de semântica usando media-types.

A base do estilo arquitetural REST é a WEB (HTTP + HTML) e para entender isso irei fazer uma analogia entre os níveis e a WEB.

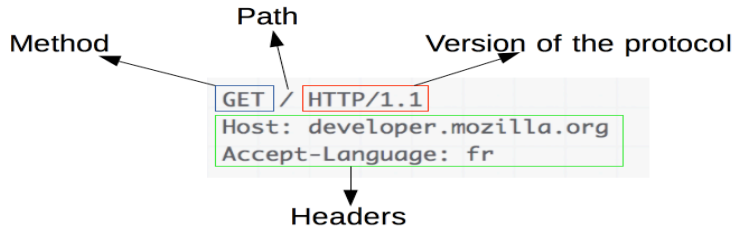
3 – PROTOCOLO HTTP

Apesar da independência de protocolo, para colocar REST na prática é necessário o uso de um protocolo, e, neste caso iremos o usar o protocolo HTPP.

HTTP é um protocolo que permite a obtenção de recursos, como documentos HTML. É a base de qualquer troca de dados na Web e um protocolo cliente-servidor, o que significa que as requisições são iniciadas pelo destinatário, geralmente um navegador da Web. Um documento completo é reconstruído a partir dos diferentes sub-documentos obtidos, como por exemplo texto, descrição do layout, imagens, vídeos, scripts e muito mais.

3.1 Requisições

Exemplo de uma requisição HTTP:

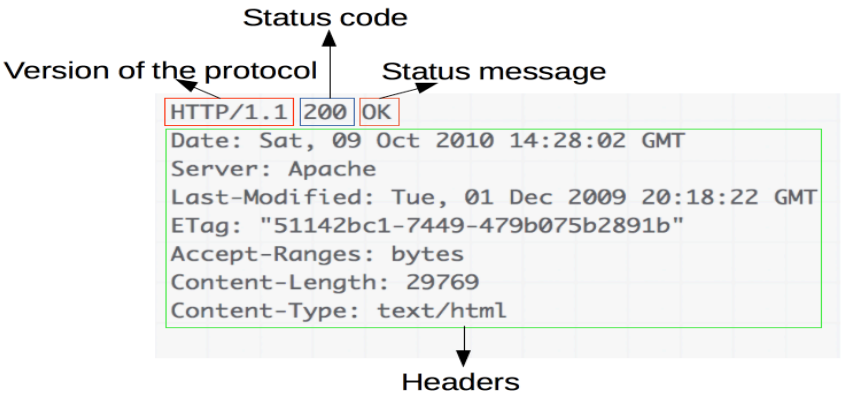


As requisições consistem dos seguintes elementos:

- Um método HTTP, geralmente é um verbo como GET, POST, DELETE, PUT, etc, ou um substantivo como OPTIONS ou HEAD que define qual operação o cliente quer fazer. Tipicamente, um cliente que busca um recurso, usa GET ou publica dados de um formulário HTML, usa POST, embora mais operações podem ser necessárias em outros casos.
- O caminho do recurso a ser buscado; a URL do recurso sem os elementos que são de contexto, por exemplo sem o protocolo (http://), o domínio (*domain*) por exemplo ufj.edu.br), ou a porta (*port*) TCP (aqui indicada pelo 80 que é ocultado por ser o número da porta padrão)
- A versão do protocolo HTTP (antes do HTTP/2.0 as mensagens eram legíveis. Com o HTTP/2.0, essas mensagens simples são encapsuladas dentro de quadros – *frames* – , tornando-as impossíveis de ler diretamente, mas o princípio se mantém o mesmo).
- Cabeçalhos opcionais que contém informações adicionais para os servidores.
- Ou um corpo de dados (*body request*), para alguns métodos como POST, similares aos corpos das respostas, que contém o recurso requisitado.

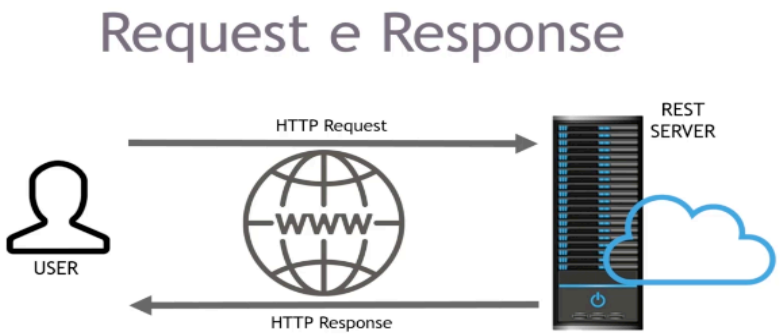
3.2 Respostas

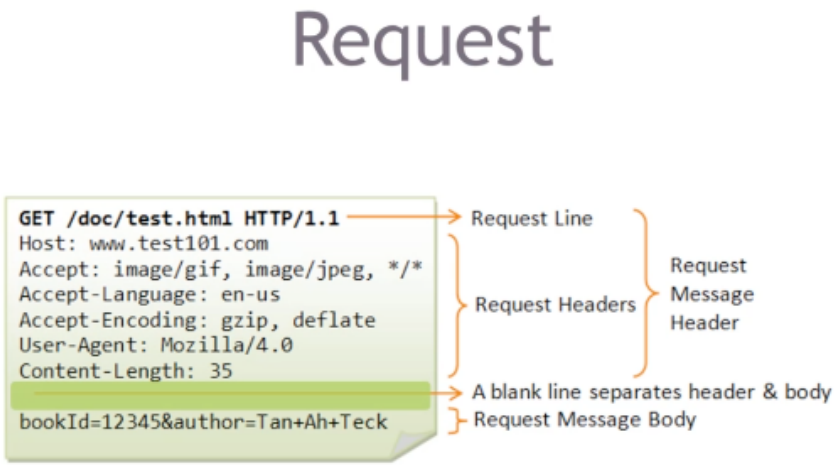
Exemplo de resposta HTTP:

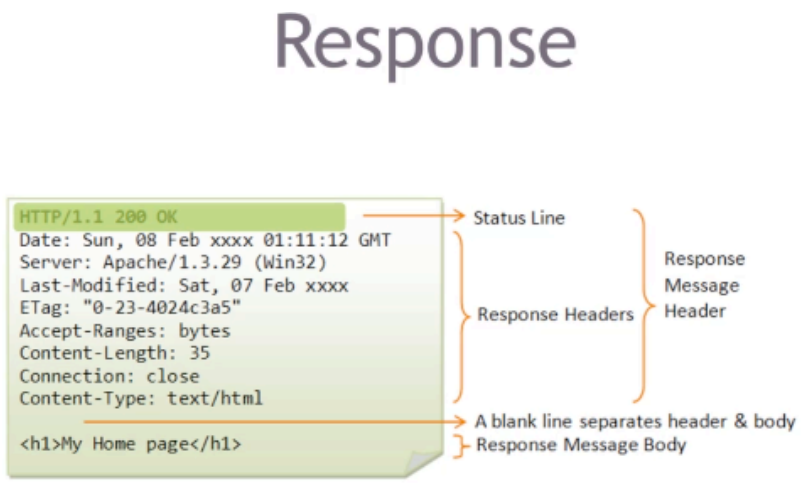


Respostas consistem dos seguintes elementos:

- A versão do protocolo HTTP que elas seguem.
- Um código de status, indicando se a requisição foi bem sucedida, ou não, e por quê.
- Uma mensagem de status, uma pequena descrição informal sobre o código de status.
- Cabeçalhos HTTP, como aqueles das requisições.
- Opcionalmente, um corpo com dados do recurso requisitado.







3.3 Identificadores, Localizadores e Nomeadores

URI ou Identificador Uniforme de Recursos ou (*Uniform Resource Identifier*) é uma cadeia de caracteres compacta usada para identificar ou denominar um recurso na Internet. O principal propósito desta identificação é permitir a interação com representações do recurso por meio de uma rede, tipicamente a Rede Mundial, usando protocolos específicos. URIs são identificados em grupos definindo uma sintaxe específica e protocolos associados. É uma forma generalizada de denominar tanto **URLs** quanto **URNs**.

Um exemplo de URI é `/caminho/recurso?querystring#fragmento`

URN ou Nome Uniforme de Recursos ou (*Uniform Resource Name*) é um tipo de URI que usa o URN Scheme e que tem por objetivo a identificação única do recurso, de forma persistente e independente da sua localização. **Não é comum a utilização do termo URN**.

Um exemplo de URN seria urn:lex:br:federal:lei:2008-06-19;11705. Esta é uma forma padronizada para referenciar esta lei. O mesmo vale para livros, metadados e outros identificadores únicos. Este identificador vive dentro de um namespace.

O que importa mais para os programadores é o URI e também a URL já que é precis identificar além de um nome único, a localização daquele recurso.

URL ou Localizador (Padrão) Uniforme de Recursos ou (*Uniform Resource Locator*) é o endereço de um recurso (como um arquivo, uma impressora etc.), disponível em uma rede; seja a Internet, ou mesmo uma rede corporativa como uma intranet.

Um URL completo tem a seguinte estrutura:

```
esquema://domínio:porta/caminho/recurso?querystring#fragmento
```

- *esquema*, ou protocolo, poderá ser HTTP, HTTPS, FTP, entre outros.
- *domínio*, ou máquina, designa o servidor que disponibiliza o documento ou recurso designado.
- *porta*, ou porto, é o ponto lógico no qual pode-se fazer a conexão com o servidor (opcional).
- *caminho* especifica o local (geralmente num sistema de arquivos) onde se encontra o - recurso dentro do servidor.
- *query string* é um conjunto de parâmetros a ser enviado ao servidor, usado para localizar, filtrar, ou mesmo criar o recurso (opcional).
- identificador de fragmento se refere a uma parte ou posição específica dentro do recurso (opcional).

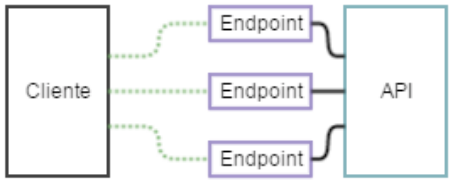
O esquema diz como conectar, o domínio especifica onde conectar, e o resto especifica o que está sendo pedido.

Exemplo:

```
http://www.ufj.edu.br/Addressing/URL/uri-spec.html
```

O protocolo é o HTTP, o servidor é designado por www.ufj.edu.br e o recurso — neste caso o arquivo (uri-spec.html)— encontra-se em Addressing/URL/. A porta, omitida, recai sobre o padrão do protocolo (no caso, 80) e não há *query string* ou identificador de fragmento.

3.4 Endpoints



Endpoint é um termo em inglês que pode ser traduzido, literalmente, como "pontos de extremidade". O termo *endpoint* pode ser aplicado a diferentes contextos. Em protocolos de comunicação, por exemplo, o *endpoint* faz referência aos terminais de conexão entre uma API e o cliente.

4 – SPRING

É um ecossistema de projetos, ou seja, um conjunto de projetos que apresentam soluções para problemas recorrentes ou caminhos para ações comuns. O objetivo deste ecossistema é permitir que o programador/desenvolvedor foque nas regras de negócio e não perca tempo com a construção de infraestrutura de código para suportar o "negócio" em si. Os projetos do Spring podem ser acessados em https://spring.io/projects.

4.1 Spring Boot

O Spring Boot é um projeto do Ecossistema Spring que facilita o processo de configuração e publicação de APIs. Ele cuida da organização e da infraestrutura necessária para que uma aplicação possa ser executada "rodar" naturalmente sem grandes esforços.

Ele consegue isso favorecendo a convenção sobre a configuração. Basta que você diga pra ele quais módulos deseja utilizar (WEB, Template, Persistência, Segurança, etc.) que ele vai reconhecer e configurar.

Você escolhe os módulos que deseja por meio dos *starters* que inclui no arquivo pom.xml do projeto. Eles, basicamente, são dependências que agrupam outras dependências. Inclusive, como temos esse grupo de dependências representadas pelo starter, nosso pom.xml acaba por ficar mais organizado.

Apesar do Spring Boot, através da convenção, já deixar tudo configurado, nada impede que você crie as suas customizações caso sejam necessárias.

Um artigo que apresenta com detalhes os conceitos de SPRING BOOT pode ser acessado em: https://blog.algaworks.com/spring-boot.

4.1.1 DevTools

DevTools é um módulo do Spring Boot que adiciona algumas ferramentas que são interessantes em tempo de desenvolvimento.

Um primeiro benefício dele é quanto a configuração de propriedades úteis para desenvolvimento.Algumas bibliotecas suportadas pelo Spring Boot usam cache para melhorar sua performance.

O segundo benefício é a reinicialização automática. Cada arquivo criado nos diretórios que fazem parte do seu *classpath* serão monitorados e qualquer alteração neles acarretará em uma reinicialização do Spring Boot. Esses diretórios são, basicamente, os diretórios `src/main/java` e `src/main/resources`.

Um último benefício que quero comentar é o fato do DevTools conter um servidor embarcado do LiveReload. O que esse servidor faz é enviar um aviso para o navegador dizendo que os nossos arquivos estáticos ou os de templates foram alterados. E com isso o navegador atualiza nossa página sozinho.

4.2 Spring Framework

O Spring é uma ferramenta na categoria Frameworks (Full Stack) de uma pilha de tecnologia. Foi desenvolvido para a plataforma Java baseado nos padrões de projetos (Design Patterns), inversão de controle e injeção de dependência.

O Spring dispõe para o programador diversas tecnologias, que simplificam o desenvolvimento de código de infraestrutura, como a Injeção de Dependência e Inversão de Controle.

Um artigo que apresenta com detalhes os conceitos de SPRING FRAMEWORK pode ser acessado em: https://enotas.com.br/blog/spring-framework/.

4.2.1 Inversão de Controle

Inversão de controle (*Inversion of Control* ou IoC) trata-se da interrupção do fluxo de execução de um código, retirando, de certa forma, o controle sobre ele e delegando-o para uma dependência ou *container*. O principal propósito é minificar o acoplamento do código.

Isso permite que exista uma facilidade na hora de trocar ou acrescentar comportamentos ao sistema (se necessário) e também diminui a possibilidade de ocorrência bugs em cascata.

No Spring Framework, a interdependência entre os objetos é mínima, ou seja, as instâncias das classes são fracamente acopladas. E a inversão de controle, no Spring, é facilitada por outro Design Pattern: Injeção de Dependência.

4.2.2 Injeção de Dependência

A injeção de dependência é um padrão de projeto já implementado pelo Spring Framework e tem como objetivo evitar o acoplamento (dependência – um objeto que uma classe depende para funcionar) de código em uma aplicação. Portanto, podemos dizer que a injeção de dependência é uma forma de aplicar a inversão de controle. E isso pode ser feito de 3 formas no Spring Framework, veja (exemplos destas formas na seção - 4.2.3 Beans - a seguir):

- anotação @Autowired;
- Construtor da Classe (Constructor Injection);
- Método Setter (Setter Injection).

Entre as três opções, a mais indicada de ser utilizada é por Construtor da Classe, isso pois traz várias vantagens, como: o aumento da legibilidade do código, facilidade de manutenção e facilidade na construção dos testes.

4.2.3 Beans

Bean é uma funcionalidade (componente) que possibilita passar a responsabilidade de gerenciamento de uma classe, e, para passar esta responsabilidade é necessário um conjunto de anotações.

@Component – anotação básica para criar qualquer tipo de ***bean*** gerencial pelo Spring Framework.

@Repository – Define um ***bean*** como sendo do tipo persistente para uso em classes e acesso a banco e dados. O Spring libera alguns recursos específicos referente a persistência de dados.

@Service – Define um classe como do tipo "serviço" (Service Layer), que possuem regras de negócio.

@Autowired – anotação usada para informar ao Spring que ele deve injetar a variável (objeto) anotada na classe em que está declarada. Veja os exemplos abaixo

```
@Service
public class AlunoService {

      @Autowired
      private AlunoDAO alunoDAO;

      public void salvarAlunos(Aluno aluno) {
            alunoDAO.save(aluno)
      }
}
```

```
@Service
public class AlunoService {


      private AlunoDAO alunoDAO;

      @Autowired
      public void setAlunoDAO(AlunoDAO alunoDAO) {
            this.alunoDAO = alunoDAO;
      }

      public void salvarAlunos(Aluno aluno) {
            alunoDAO.save(aluno)
      }
}
```

```
@Service
public class AlunoService {


      private AlunoDAO alunoDAO;
```

```
        @Autowired
        public AlunoService(AlunoDAO alunoDAO) {
              this.alunoDAO = alunoDAO;
        }


        public void salvarAlunos(Aluno aluno) {
              alunoDAO.save(aluno)
        }
  }
```

5 – CRUD

CRUD é o acrônimo da expressão do idioma Inglês, *Create* (Criação), *Read* (Consulta), *Update* (Atualização) e *Delete* (Exclusão). Este acrônimo é comumente utilizado para definir as quatro operações básicas usadas em Banco de Dados Relacionais. Outros acrônimos podem ser usados para definir as mesmas operações: ABCD: *Add, Browse, Change and Delete* BREAD: *Browse, Read, Edit, Add and Delete* e VADE: *View, Add, Delete, Edit.*

Em ambientes REST, CRUD frequentemente corresponde aos métodos (verbos) HTTP POST, GET, PUT e DELETE.

Um artigo que apresenta com detalhes os conceitos de CRUD pode ser acessado em: https://www.codecademy.com/articles/what-is-crud.

Também é necessário neste momento fazer uma relação de CRUD com DAO e DTO (*Data Access Object e Data Transfer Object* respectivamente). Em geral na arquitetura MVC (Model-View-Control) o CRUD representa a camada entre o Modelo e a Visão, sendo manipulada pelo Controle.

6 – O padrão MVC (Movel-View-Controller)

O MVC é um modelo arquitetural utilizado em muitos projetos devido a estrutura que possui, o que possibilita a divisão do projeto em camadas muito bem definidas. Cada uma delas, o Model, o Controller e a View, executa o que lhe é definido e nada mais do que isso.

A utilização do padrão MVC traz como benefício o isolamento das regras de negócios da lógica de apresentação, que é a interface com o usuário. Isto possibilita a existência de várias interfaces com o usuário que podem ser modificadas sem a necessidade de alterar as regras de negócios, proporcionando muito mais flexibilidade e oportunidades de reuso das classes.

Uma das características de um padrão de projeto é poder aplicá-lo em sistemas distintos. O padrão MVC pode ser utilizado em vários tipos de projetos como, por exemplo, desktop, web e mobile.

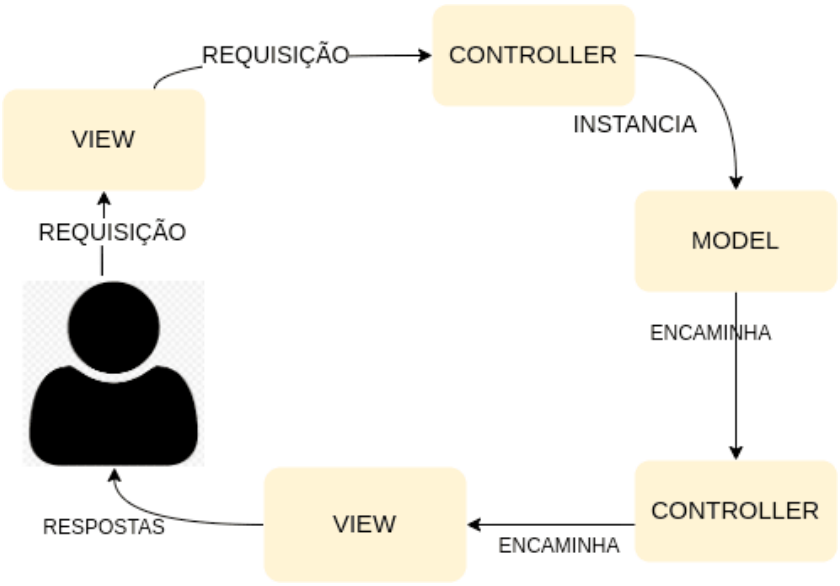
A comunicação entre interfaces e regras de negócios é definida através de um controlador, que separa as camadas. Quando um evento é executado na interface gráfica, como um clique em um botão, a interface se comunicará com o controlador, que por sua vez se comunica com as regras de negócios.

Imagine uma aplicação financeira que realiza cálculos de diversos tipos, como os de juros. Você pode inserir valores para os cálculos e também escolher que tipo de cálculo será realizado. Isto tudo é feito pela interface gráfica, que para o modelo MVC é conhecida como View. No entanto, o sistema precisa saber que você está requisitando um cálculo, e para isso, terá um botão no sistema que quando clicado gera um evento.

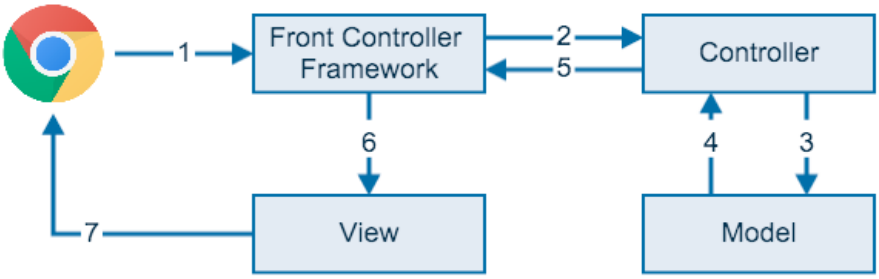
Este evento pode ser uma requisição para um tipo de cálculo específico como o de juros simples ou juros compostos. Fazem parte da requisição os valores digitados no formulário e a seleção do tipo de cálculo que o usuário quer executar sobre o valor informado. O evento do botão é como um pedido a um intermediador (Controller) que prepara as informações para então enviá-las para o cálculo. O controlador é o único no sistema que conhece o responsável pela execução do cálculo, neste caso, a camada que contém as regras de negócios. Esta operação matemática será realizada pelo Model assim que ele receber um pedido do Controller.

O Model realiza a operação matemática e retorna o valor calculado para o Controller, que também é o único que possui conhecimento da existência da camada de visualização. Tendo o valor em mãos, o intermediador o repassa para a interface gráfica que exibirá para o usuário. Caso esta operação deva ser registrada em uma base de dados, o Model se encarrega também desta tarefa.

Um artigo que apresenta com detalhes os conceitos de MVC pode ser acessado em: https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308.



6.1 Spring MVC



Detalhadamente, os passos mostrados na imagem acima, são:

1. Acessamos uma URL no browser que envia a requisição HTTP para o servidor que roda a aplicação web com Spring MVC. Perceba que quem recebe a requisição é o controlador do framework, o Spring MVC.

2. O controlador do framework irá procurar qual classe é responsável por tratar essa requisição, entregando a ela os dados enviados pelo browser. Essa classe faz o papel do *controller*.

3. O *controller* passa os dados para o *model*, que por sua vez executa todas as regras de negócio, como cálculos, validações e acesso ao banco de dados.

4. O resultado das operações realizadas pelo *model* é retornado ao *controller*.

5. O controller retorna o nome da view, junto com os dados que ela precisa para renderizar a página.

6. O Framework encontra a view que processa os dados, transformando o resultado em um HTML.

7. Finalmente, o HTML é retornado ao browser do usuário.

Na prática, o "*controller*" é a classe Java com os métodos que tratam essas requisições. Portanto, tem acesso a toda informação relacionada a ela como parâmetros da URL, dados submetidos através de um formulário, cabeçalhos HTTP, etc.

Um artigo que apresenta com detalhes os conceitos de Spring MVC pode ser acessado em: https://blog.algaworks.com/spring-mvc/.

6.2 – Anotações Spring MVC

@Controller (usado para configurar a classe em um *bean* do tipo *controller* do MVC).

@RequestMapping (usado para mapear URLs de acesso a um *controller* e aos métodos contidos nele. Também pode-se definir verbos HTTP (POST, GET, PUT, DELETE e outros) de acesso aos métodos específicos criados dentro da classe definida como @Controller. Veja o exemplo abaixo com o uso das duas anotações Spring MVC

```
@Controller
@RequestMapping("/auladaw")
public class AlunoController {

	@RequestMapping(path = "/listar", method = RequestMethod.GET)
	public String buscaAlunos() {
	}
}
```

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping (usados para especificar diretamente qual será o método de requisição. Possuem enquadramento com as ações do CRUD (Create, Read, Update e Delete). Estes representam apenas os principais métodos de requisição (verbos HTTP). O exemplo abaixo mostra o código anterior com o uso específico da anotação @GetMapping.

```
@Controller
@RequestMapping("/auladaw")
public class AlunoController {

	@GetMapping("/listar")
	public String buscaAlunos() {
	}
}
```

Para ilustrar este funcionamento desta estrutura de código, em um cliente como um navegador (web browser), e, desde que a aplicação em que este código esteja presente, esteja ativa (foi compilado e executado juntamente com um *container* como o Tomcat) digitará no campo de endereço: http://localhost/8080/auladaw/listar e terá como retorno, desde que implementado, pois o método "buscaAlunos()" está vazio, a lista de alunos, por exemplo, de uma base qualquer de dados.

@PathVariable (usado para extrair da requisição um parâmetro que foi incluído como *path* da URL, como mostra o exemplo abaixo, tendo como base a seguinte url: http://localhost/8080/auladaw/listar/2).

```
@Controller
@RequestMapping("/auladaw")
public class AlunoController {

	@GetMapping("/listar/{id}")
	public String buscaUmAluno(@PathVariable("id") Integer id) {
	}
}
```

Neste caso, o parâmetro "id" que está entre chaves no @GetMapping será responsável por receber um valor (inteiro neste caso) e retornar uma resposta usando o valor.

@RequestParam (usado para capturar um parâmetro de consulta enviado na solicitação, como mostra o exemplo abaixo, tendo como base a seguinte url: http://localhost/8080/auladaw/listar?id=2).

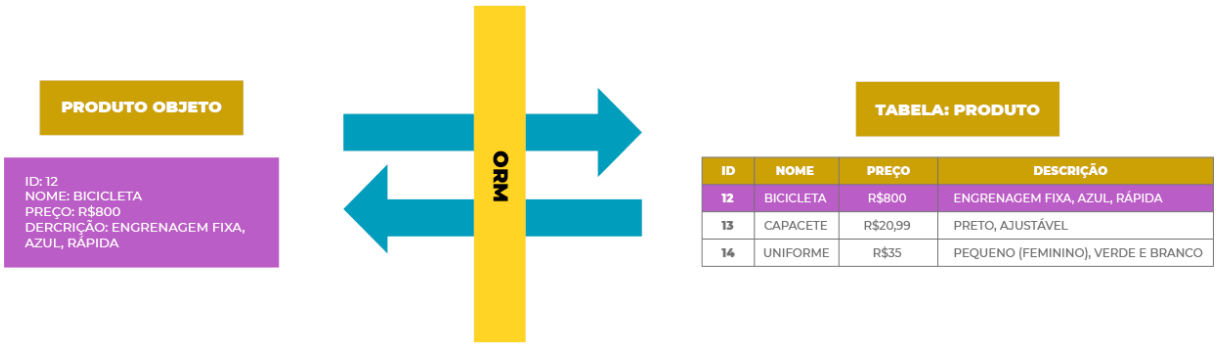
```
@Controller
@RequestMapping("/auladaw")
public class AlunoController {
```

```
        @GetMapping(“/listar”)
        public String buscaUmAluno(@RequestParam(name = “id”)  Integer id) {
        }
}
```

@Valid (anotação responsável por injetar uma validação back-end por meio do Bean Validation, Spring Validator ou Hibernate Validator).

7 – ORM - Object-Relational Mapping (Mapeamento Objeto-relacional)

ORM define uma técnica para realizar a conciliação entre o modelo Entidade-Relacionamento (Dados) e Classes-Objeto (Aplicação). Uma das partes centrais é através do mapeamento de linhas para objetos:

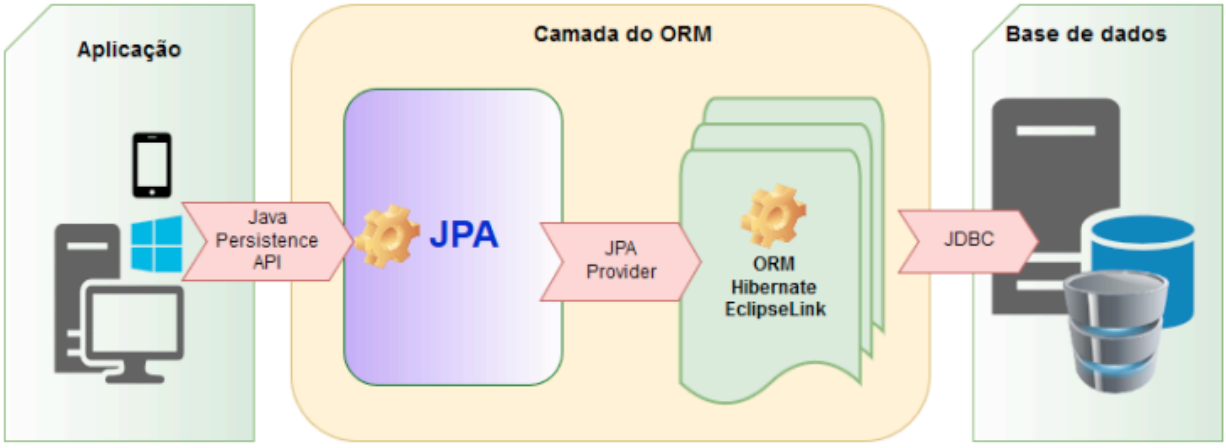


Os principais ORMs do mercado para linguagem Java são: Hibernate; EclipseLink; e, ActiveJPA.

Um artigo que apresenta com detalhes os conceitos de ORM pode ser acessado em: https://www.treinaweb.com.br/blog/o-que-e-orm/**.**

8 – JPA – Spring Data JPA

Java/Jakarta Persistence Application - JPA é a especificação do Java que dita como os frameworks ORM devem ser implementados. Ela foi criada com o intuito de padronizar essas soluções. Antes de sua criação existiam diversos frameworks e bibliotecas que abstraiam os desafios da persistência com ORM em Java.



Um artigo que apresenta com detalhes os conceitos de JPA pode ser acessado em: https://notasecodigos.wordpress.com/2017/05/06/persistencia-de-dados-com-jpa.

9 – JSON

JSON (JavaScript Object Notation) é um formato leve de troca de informações/dados entre sistemas ou aplicações (comum em comunicações *backend e frontend*). O JSON além de ser um formato leve para troca de dados é também muito simples de ler. Além do formato JSON, existem outros que podem ser usados na troca de dados entre uma aplicações

Um artigo que apresenta com detalhes os conceitos de JSON pode ser acessado em: https://www.devmedia.com.br/o-que-e-json/23166.