—----------Spring----------

**Spring Framework** : 1 É uma framework aplicacional, de inversão de controle (IoC) e injeção de dependência para a plataforma Java. 2 Oferece uma variedade de recursos para ajudar os desenvolvedores a construir aplicativos robustos e escaláveis,incluindo gerenciamento de transações, segurança, gerenciamento de dados e suporte para múltiplos protocolos de rede. 3 As funcionalidades da framework podem ser usadas por qualquer aplicação Java, mas existem extensões para a construção de aplicações web,em cima da plataforma Java EE.

**Spring Boot** - ( Ou Java Spring Boot) é uma ferramenta que torna o desenvolvimento de aplicações web e microserviços com a framework Spring, mais rápido e fácil através de 3 capacidades fundamentais: .Autoconfiguração. .Uma abordagem opinativa para configuração. .A capacidade de criar aplicativos autônomos. O Spring Boot ajuda a criar aplicativos que não estão vinculados a uma plataforma específica e que podem ser executados localmente em um dispositivo sem conexão à internet ou outros serviços instalados para funcionar de forma adequada.

**Spring Initializr**: É um serviço de web que gera uma estrutura de projecto (project structure) para uma nova aplicação Spring Boot.
Gera: Projecto JVM Gradle-Based Bibliotecas incluidas: SPring, jackson, Kotlin etc , Função main e a anotação @SpringBootApplication

## *1.1 Spring Context*

**Spring Context** - É um container que instância, configura, e gerencia uma conjunto de objectos (Beans) que são definidos com metadados de configuração. .Dependency Injection (DI) container: Dado uma série de tipo de informação, incluindo as suas dependências respectivas, cria um grafo de instâncias de componentes.Sistema que nos permite gerenciar a composição e o ciclo de vida dos objetos e suas dependências, dinamicamente. .O contexto é o componente Spring responsável por instanciar e gerenciar instâncias, como instâncias de controladores. **Bean:** Objecto que é criado e gerenciado pelo Context, pode ser criado através da anotação @Bean ou pela criação de um context e registando um componente (Bean) nele. O Spring automaticamente regista qualquer componente com anotações do tipo @Component, @Service etc como Beans. @Component é auto-configurado pelo Spring automaticamente, enquanto @Bean declara explicitamente uma configuração

```
@Configuration
class BeanConfig {
    @Bean
    fun httpClient(cookieHandler: CookieHandler): HttpClie
        .newBuilder()
        .cookieHandler(cookieHandler)
        .build()
}

// Create the context
val context = AnnotationConfigApplicationContext()

// Add the bean definitions
context.register(BeanConfig::class.java)
```

**Dependency Injection (DI) e Inversion of Control (IoC)**

1. .Criação do gráfico de objetos da aplicação.
2. .Instanciação dos objetos e suas dependências, passando as dependências para os objetos.
3. .O objeto dependente recebe e usa as dependências, mas não as cria, quem cria, e gerencia, é o IoC (que faz dependency Injection)

Handler (dependente) ------Usa---> Service (Dependência)

**Constructor injection**:A inversão de controle é usada para injeção de dependência através do construtor, que é a maneira preferencial de injeção de dependência;

**Field/Property injection:** Injeção de dependências através de campos/propriedades, onde as dependências são passadas após a criação do objeto, através da atribuição de campos ou execução de métodos setter de propriedade.

1. Requer campos de dependência mutáveis.
2. Os objetos não estão prontos para operar após a construção.
3. Usado apenas quando a Construction Injection não é possível.

```
@RestController
@RequestMapping("/status")
class getProcessingTime {
        private val processingTimes = mutableMapOf<String,ProcessingTime>() // Tempo em milisegundos

@GetMapping("/{method}")
fun getProcessingTime(@PathVariable method : String,
        if(processingTimes.get(method)==null) return ResponseEntity.notFound().build()
        else return ResponseEntity.ok(processingTimes.get(method))
}

@PostMapping("/{method}")
fun updateProcessingTime(@PathVariable method : String, @RequestBody min : String, @RequestBody max : String){
        processingTimes.put(method,ProcessingTime(min,max))
}

data class ProcessingTime(val min: Int, val max: Int)
```

**Wiring:** Estabelece as dependências entre os objectos.

**Class Stereotype:** Containers de funcionalidade (Components) vs Containers de Data.

***Exemplo:*** Passos:

1. Criação do Context
2. Adicionar as definições de Bean ao Context.
3. Fazer Refresh ao Context, para ter em consideração as novas definições de Bean do passo anterior. 4. Usar o Context para ir buscar os Beans.

```
// Create the context
val context = AnnotationConfigApplicationContext()

// Add the bean definitions
context.register(
    ComponentA::class.java,
    ComponentB::class.java,
    ComponentC::class.java
)

// Refresh the context
context.refresh()

// Get the beans (example: ComponentB)
val componentB = context.getBean<ComponentB>()
```

**BeanFactory**: Core container Interface .ApplicationContext: A interface que extende de BeanFactory .AnnotationConfigApplicationContext: A class que implementa AbstractApplicationContext e é usada para criar o Context através das anotações.

**O Spring Context é um DI** (Dependency Injection) container que precisa de metadata de configuração para criar os objectos da aplicação.
A metadata pode ser passada de várias maneiras: .Ficheiros de configuração XML ou .Anotações Java ou código Java

**Configuração de Container através de Anotação**:.A anotação @Configuration indica que a classe declara um ou mais métodos @Bean e pode ser processada pelo Spring .A anotação @Bean indica ao Spring que um método anotado com @Bean retornará um objecto que deve ser registado como Bean no contexto da aplicação Spring..A anotação @Autowired é usada para fazer wire automaticamente do Bean no método Setter, no construtor, ou numa propriedade..A anotação @Component é um estereótipo genérico para qualquer componente gerido pelo Spring: -A anotação **@Service** é uma especialização da anotação @Component para uma camada de service. -A anotação **@Repository** é uma especialização da anotação @Component para uma camada de presistence.-A anotação **@Controller** é uma especialização da anotação @Component para uma camada de presentation.

- A camada de persistência em Spring é responsável por lidar com o armazenamento e recuperação de dados, geralmente utilizando o framework de persistência Hibernate ou JPA. Ela é responsável por realizar operações como salvar, atualizar, excluir e buscar dados no banco de dados.
- A camada de serviço em Spring é responsável por implementar as regras de negócio e lógica de aplicação. Ela é responsável por realizar tarefas específicas e complexas, como validações, cálculos e processamento de dados.
- A camada de apresentação em Spring é responsável por lidar com a interface do usuário e comunicação com o cliente. Ela é responsável por controlar o fluxo de navegação, receber e enviar dados para o usuário e geralmente é implementada utilizando o framework Spring MVC.

## *1.2 - Spring MVC*

**O Spring MVC** é uma biblioteca/framework para http request handling, fornece funcionalidades de alto nivel no topo da Java Servlet API. .É baseada no padrão Model-View-Controller (MVC).

**Servlet**: .Um Servlet é uma classe de Java que é usada para extender as capacidades dos servidores que dão host a aplicações acedidas por meios de um modelo request-response.
.Permitem que request handlers (servlets) e intermediários (filters) corram em múltiplos web servers, os chamados de servlet containers.
.Final handler para um request HTTP:
.HttpServletRequest - Representa a informação do Request.
.HttpServletResponse - Representa a informação produzida até ao momento da Response.
.Ambos os HttpServletRequest e HttpServletResponse são únicos para cada par request-response.

**Filtro**: .Intermediário que pode fazer:
.Pre-Processing: Antes do par request-response ser passado para o próximo intermediário ou final handler.
.Post-Processing: Depois do próximo intermediário ou final handler retornar.
.Short-Circuit do processing do pedido ao não chamar o próximo intermediário ou final handler.

```
RequestParam é usado para recuperar
valores por query string,
que são enviados como parte da URL, geralmente
após o símbolo "?".
PathVariable é usado para recuperar valores de
parâmetros que são enviados incluídos diretamente na URL,
 geralmente em uma estrutura de caminho específica,
como "/users/{id}".
```

**Design**: .Um Dispatch Servlet que é registado no topo de um certo servlet server, para todas as rotas abaixo da rota base ("/"). .Request Handlers que são responsáveis pelo processamento de requests para especificas rotas - uma maneira de definir handlers é através de instâncias de métodos de uma classe Controller

| MÉTODO | IDEMPOTÊNTE | SEGURO |
|---|---|---|
| GET | Sim | Sim |
| POST | Não | Não |
| PUT | Sim | Não |
| DELETE | Sim | Não |
| HEAD | Sim | Sim |
| OPTIONS | Sim | Sim |

**Argument Binding**:
.Mecanismo pelo qual o Spring MVC computa os argumentos para passar aos handlers
.Um Argument Resolver é um componente que é responsável por resolver um parâmetro de um método para um argument value, de um certo pedido.
.Um Controller é uma anotação de classe com @Controller ou @RestController que contém funções de request handling
.Um controller é um singleton por omissão, isto significa que a mesma instância é usada para todos os pedidos.
.Isto acontece porque o Spring MVC segue um padrão Singleton, para assegurar que apenas uma instância de um objecto existe por aplicação.
.Um Handler method, ou função Handler, é uma instância dentro de uma classe Controller, anotada com @RequestMapping ou @GetMapping, @PostMapping, etc.
.O mesmo handler pode ser chamado concorrentemente por múltiplas threads, pois cada request é handled por uma thread diferente.

**@Controller vs @RestController**: A anotação @RestController é uma especialização da anotação @Controller, com o comportamento adicionado de que o valor do retorno de cada função deste Controller é automaticamente serializado para JSON e passado de volta para o objecto HttpServletResponse. Combina o comportamento do @ResponseBody e o @Controller. É bastante útil na construção de RESTful APIs.

**Servlet Filters:** .Mecanismo definido pela API Java Servlet; .Permitem que o código seja executado antes e depois do processamento da solicitação pelo servlet; .Eles também podem curto-circuitar o processamento da solicitação, finalizando o ciclo request-response; .Eles não têm acesso a informações específicas do Spring, como o handler que será executado.

**Handler Interceptors:** .Mecanismo específico do Spring para interceptar a chamada ao handler do request, ou seja, ter código executado antes e depois da execução do handler; .Eles têm acesso às informações específicas do handler .Desvantagem: Eles são executados mais tarde no pipeline, depois do dispatch servlet. É possível configurar interceptors para serem criados por cada requisição, ou para serem compartilhados entre várias requisições. Criados de varias maneiras. Os métodos dos interceptores não precisam ser estáticos.

**Exception Handling:** .A anotação @ControllerAdvice permite definir uma classe que será usada para lidar com exceções lançadas pelos manipuladores; .Há duas maneiras de representar erros em uma API: .Baseado em Exceção:.O handler lança uma exceção; .A exceção é tratada por uma classe com anotação @ControllerAdvice; .A exceção é traduzida em uma resposta HTTP; .Baseado em Retorno:.O handler retorna um objeto ResponseEntity; .O objeto ResponseEntity é traduzido em uma resposta HTTP..

---

## *Web APIs*

---

Uma Web API é uma série de funções e procedimentos que permitem a criação de aplicações que acedem as funcionalidades ou os dados de um sistema operativo, aplicação, ou outro serviço

**System Architecture**

**Backend:**.Serviço Responsável por assegurar a integridade dos dados, especialmente para que todas as regras de domain sejam respeitadas: lógica de negócio .Server side
**Frontend**:.Responsável pela interação do usuário .Client-Side
A interação entre o frontend e o backend é feita através de uma API HTTP, fornecida pelo serviço backend. .A iniciativa de comunicação é sempre feita na API do cliente. .O serviço backend só comunica informação para o serviço frontend na forma de uma response HTTP. .Implica polling pela aplicação frontend para verificar mudanças de estado asincrono.
Arquitetura do serviço Backend: .Uma DBMS (Database Management System), com o estado e dados do sistema .Um ou mais servidores, a correr uma aplicação, hosting a lógica de requests HTTP, a lógica de domínio, e a lógica de acesso a dados. .Load Balancer, para distribuir os pedidos por multiplos servidores.
**Handler**: .Sabe como dar handle (lidar) um pedido (request). .Sabe como serializar e deserializar o request e o response .Sabe HTTP .NÃO sabe a lógica de negócio
**Serviço:** .Contém a lógica de negócio .NÃO sabe HTTP .Não sabe como interagir com a Database
**Repository**: .Sabe como interagir com a Database .Não sabe a lógica de negócio
**Data Access(DBMS):**´ .Sabe como interagir com a Database .Nºao sabe a lógica de negócio

---

## Web Architecture

---

**Siren:** .Uma especificação hypermedia para representar entidades de maneira fáci para navegação para estas ou ações relacionadasl | .Inclui representações de ações em representações de recursos: actions property | .Agrupando as propriedades de recursos non-linked em: properties property | .A propriedade links é usada para representar links navegacionais para outros recursos, páginas incluem sempre um link referência para si mesma. | .A estrutura do link inclui uma propriedade "href"que especifica o URI para o linked resource. | O atributo rel que contém o tipo da relação entre dois recursos, eventualmente representado como link | Link também contém atributos como “title” “type” e “class” .Estrutura do header Links: "links": [ { "rel": [ "self" ], "href": "http://api.x.io/customers/pj123" } ] | Siren é .Media ty pe: application/vnd.siren+json
[*Nota: links vs actions -* A diferença entre links e actions é que os links são navigational e as ações são state-changing. Links são usados para navegar para outros recursos, enquanto as actions são usadas para mudar o estado do recurso atual. Noutras palavras os links são usados para navegar para outros recursos, enquanto as actions são usadas para realizar operações no recurso atual. O elemento <Link/> é conhecido pelo React Router e é usado para navegação para uma página que será rendered no client-side, não sendo feito nenhum pedido ao servidor. Enquanto o elemento <a/> é usado para a criação de hyperlinks
**Problem Details for HTTP APIs**: .Um tipo de mídia para JSON que fornece uma maneira padronizada de representar respostas de erro em respostas JSON;.Um objeto de problem details contém as seguintes propriedades:.tipo: URI que identifica o tipo de problema; .título: resumo curto e legível por humanos do tipo de problema; .status: o código de status HTTP gerado pelo servidor de origem para esta ocorrência do problema; .detalhe: uma explicação legível por humanos específica para esta ocorrência do problema; .instância: uma referência URI que identifica a ocorrência específica do problema;.Tipo de mídia: application/problem+json.
**intermediários HTTP**:.Proxy .Gateway (AKA Reverse-Proxy) .Tunnel

---

## The Browser Application Platform

---

.Um **módulo** é um mecanismo para organizar e isolar o código .Por default, todo o código que executa num browser corre num top-level scope, chamado de global scope (múltiplos script elements existem no mesmo scope) .Node.js usa o CommonJS(CJS) module system .Browser usa o ES6 (ESM, ECMAscript) module system
**ESM**:.O module system ESM está built-in no browser. .Para importar um módulo, usa-se o statement "import" .Um module é um mecanismo para organizar e isolar código, contudo se a aplicação contém muitos modules, o browser necessita de carregar muitos ficheiros, o que irá impactar o performance da aplicação.Porém, não se deve reduzir o numero de modules para questões de eficiência. A solução é compactar múltiplos módulos num único load level modules..Isto Significa que um processo de build é necessário para fazer bundle dos modules para um único ficheiro.
**Webpack:**.O Webpack é um module bundler. O seu propósito é fazer bundle de ficheiros JavaScript para uso num browser, também é capaz de transformar, bundlar, ou empacotar praticamente qualquer recurso. .O ficheiro de configuração webpack é o webpack.config.js .O webpack também contém um development server que pode ser usado para servir a aplicação .Por default, o webpack irá fazer bundle de todos os modules na pasta "src" num único ficheiro na pasta "dist" chamado de "main.js". Permite a utilização de recursos como importação e exportação de módulos, que são nativos do Node.js (ou do CommonJS), mas não suportados nativamente pelo browser. Ele é uma das ferramentas mais comuns utilizadas para garantir a compatibilidade dos módulos do NPM com o browser. o Gradle também permite a inclusão de bibliotecas do NPM no projeto Spring, é mais simples e flexível, é usado no Backend, Webpack no frontend. Contém loaders que delegam para por exemplo o compilador typescript, neste caso o ts-loader, para converter código tsx em js.

---

## React

---

**React** é uma biblioteca JavaScript que manipula trees que representam interfaces de usuário (UIs), incluindo o DOM tree do browser
.A UI para uma browser window é definida pelos conteúdos do DOM tree (Document Object Model). .Esta tree é composta por DOM nodes, que incluem element nodes ou text nodes.
**Declarative** .React é declarativo, ou seja o developer define o UI ao descrever a tree .Não se manipula o DOM diretamente, mas sim o virtual DOM .O virtual DOM é uma tree que é guardada em memória e é atualizada quando o UI muda .O React então compara a virtual DOM com a verdadeira DOM tree e faz update da verdadeira DOM apenas quando necessário .Os elementos da virtual tree nunca são mutados após serem criados -> mudar o UI é feito através da criação de uma nova virtual tree
**Component-Based** O React é component-based, ou seja, define-se a UI através de componentes. .Um componente é uma função que retorna uma tree
**React packages**: .React, é um host-dependent package, e sabe como manipular a DOM tree, ou qualquer outra tree como Android View tree - é o core do React e é chamado de Renderer. .React-Dom é um host-independent package, e é usado para criar tree representations, antes que sejam aplicadas para uma especifica host tree, e é chamado de concilier (reconciliation é o processo de dar update à DOM tree para dar match (ficar igual) à virtual DOM tree.
**React Components:** .Um componente é uma função que implementa o método render(), que recebe propriedades como input, e retorna uma tree como output. .Usa a sintaxe JSX, permite escrita de código XML-like (parecido ao html) dentro do JavaScript. .O Babel transforma JSX em React.createElement(). **Render**: Chamada à função que define o componente.
  Note the use of a component bellow  <MyComponent awesome="of-course"/> 
Também pode ser escrito como: React.createElement('div', null , React.createElement('p', null, 'Note The Use of a Component bellow'), React.createElement(MyComponent, {awesome: 'of-course'})) React.createElement(type,props, [..children]). .type: é uma string (div, span, etc) ou um componente .props: é um objecto que contem as propriedades do elemento, ou null .children: são os filhos do elemento (ou seja outros React elements) .Esta função não pode ser confundida com document.createElement function, which returns a DOM element, not a React element (object that represents a DOM element) - this happens because React operates a Virtual DOM.
**Lifecylce:** .Mounting é o processo de inserir um componente no DOM, como render() ou componentDidMount() .Unmounting é o processocontrário,, com componentWillUnmount()
**Hooks**: .Hooks são funções que permitem que se fixe funcionalidades de funções de componentes no React state e nas funcionalidades do lifecycle. .Hooks não funcionam dentro de classes - eles deixam que se usem React sem classes .Nunca se pode chamar Hooks dentro de loops, condições ou funções aninhadas, pois estes são dependentes da ordem em que são chamados devem ser sempre usados no top level de uma função React .Assim assegura-se que são chamados antes de um return, e por mesma ordem desejada cada vez que um componente faz render, tendo acesso ao scope correto. .Apenas se pode chamar Hooks através de um component, não de funções regulares JavaScrit .useState hook: permite que se adicione estado React a function components. .useEffect hook: permite que se faça side effects numa function component. .useReducer hook: é uma alternativa ao useState.
**React Context**: fornece uma maneira de passar dados pela component tree sem ter de passar props manualmente a cada nivel da tree. `setMyArray(oldArray => [...oldArray, newElement]);`
**React Router:** .History: Quando o suer navega, o browser acompanha a navegação e mete num stack cada uma.
**Path Templates**:.O elemento <Route/> renderiza um componente associado a uma certa rota na propriedade "path", quando o user navega para esta.

```
const [urlData, setUrlData] = useState([])

useEffect(() => {
    fetchUrls()
}, [props.urls])

const fetchUrls = () => {
    const { urls } = props
    const urlData = []
    urls.forEach((url, i) => {
        fetch(url)
            .then(response => {
                urlData[i] = {
                    url,
                    status: response.status,
                }
                setUrlData(urlData)
            })
            .catch(error => {
                urlData[i] = {
                    url,
                    error,
                }
                setUrlData(urlData)
```

```
import React, { useState, useEffect } from 'react';
const MessageSlider = ({ messages, period, component: Component }) => {
    const [currentMessage, setCurrentMessage] = useState(0);
    useEffect(() => {
        const interval = setInterval(() => {
            if (currentMessage === messages.length - 1) {
                setCurrentMessage(0);
            } else {
                setCurrentMessage(currentMessage + 1);
            }
        }, period);
        return () => clearInterval(interval);
    }, [currentMessage, messages, period]);
    return <Component text={messages[currentMessage]} />;
};
```

```
@RestController
@RequestMapping("/pending")
class PendingController {
    private val pendingRequests = mutableMapOf<string,Int>()

@GetMapping
fun getPendingRequests():ResponseEntity<Map<string,Int>>{
        return ResponseEntity.ok(pendingRequests)
}

@PostMapping
fun addPendingRequest(@RequestBody method:string){
        pendingRequests[method] = pendingRequests.getOrDefault(method,0) + 1
}

@PutMapping
  fun updatePendingRequest(@RequestBody method:string){
        pendingRequests[method] = pendingRequests.getOrDefault(method,0) - 1
```

Deep linking vai sair. O que é neccesário realizar para uma SPA suporta deep-linking? (Atenção, deep-linking pode ser aplicável tanto ao frontend como ao backend, mas neste caso como se pede uma SPA, é frontend, caso fosse backend era retornar um elemento HTML para uma certa rota pedida)

.A Aplicação necessita ter algo que sabe apresentar conteúdos em relação a um certo caminho (Dado URI, mostrar uma UI associada, como o React Router)

window.location é uma função do browser, não do React.
window.location.pathname faz pedido à rota em que o browser vai passar a estar, mas o getHistory não.
O interceptor não tem acesso ao Spring Context.
O Interceptor tem acesso a informação sobre o handler que vai processar o pedido que vai ser interceptado.
Os filtros não tem essa informação, apenas à informação do pedido e à da resposta.
É suposto sabermos o que são handlers, e que o interceptor tem acesso ao handler, e que vai lidar com o pedido que vai ser interceptado.
Handler: Métodos em uma classe Controler.
O Handler method é um objecto que tem informação sobre o Handler. Permite perguntar ao handler em que controller é que está, que anotações é que tem etc.
As instâncias não são partilhadas.
Pedido anonimo - Um pedido que não tem informação que possa verificar a identidade do emissor (não tem autenticação).
Não ter token, verificar o header.