Main differences between links and actions:
Links:
-Represent navigational transitions.
-Has rel attribute but doesn't have method.
-Identifies a resource that is related to the current context.
-Method is always GET.

Actions:
-Show available behaviors an entity exposes.
-Has method attribute but doesn't have rel.
-Normally used to change the state of the current context.
-Normally uses Method that aren't GET.
-Has all the information needed to perform the action.

O uso de **Hypermedia** nas representações dos recursos de uma API HTTP promove uma melhor organização, compreensão e facilidade de navegação por parte dos clientes dessa API e também proporciona uma maior flexibilidade na manutenção e evolução da API ao longo do tempo.

Servlet Filter:
É usado para interceptar e processar pedidos antes de enviá-las ao dispatcher servlet.
Manipular pedidos e respostas antes e depois de serem processados pelo dispatcher servlet.

Handler Interceptor:
É usado para interceptar e processar solicitações antes de enviá-las aos controllers.
Podem ser usados para pré e pós processamento de pedidos.
Pode ser associado a controllers específicos não interferindo com os restantes.
Tem uma fácil configuração no Spring MVC.

A estrutura do *header* **Link** consiste no URI do recurso relacionado e no identificador da relação do recurso representado e do alvo(rel). Adicionalmente, pode ter 0 ou mais atributos adicionais. O propósito do *header* **Link** é fornecer ao cliente informação sobre o recurso relacionado colocando essa informação no header e não no body.

A razão para esta limitação deve-se a necessidade de garantir que os Hooks sejam chamados de uma maneira consistente e que haja uma garantia que a execução dos hooks é única por renderização do componente. Uma vez que, a falta dessas garantias, poderia levar a comportamentos inesperados e difíceis de compreender e a ineficiências dos componentes.

A consequência de anotar classes com a anotação **@Component** é registar essas mesmas classes como componentes Spring, ou seja, quando a aplicação for executada,o contentor de injeções de dependências(context) vai procurar as classes anotadas com **@Component** para instaciá-las(criar beans) e consequentemente injetar essas dependências nas classes que necessitam delas (classes que têm um parâmetro no seu construtor do tipo das classes anotadas com **@Component**). Neste processo estão envolvidos conceitos como: Dependência (ex: class A que depende da classe B), injeção de dependência (a colocação da instância da classe anotada na classe que depende desta) e inversão de controlo (A classe que depende de outra não cria uma instância, mas sim recebe uma no seu construtor).

Headers:
Content-Location: Pedido e resposta
Cache-Control: Pedido e resposta
Authorization: Pedido
Link: Pedido e resposta
Location: Resposta
Last-Modified: Resposta

Segundo o RFC 8288 (Web Linking), um link relation type é o identificador que expressa a semântica da ligação entre dois recursos.

A relação entre negociação de conteúdo e caching, no contexto do protocolo HTTP, está relacionada à gestão eficiente de versões diferentes de um recurso e à entrega apropriada dessas versões aos clientes. Isto acontece porque, a negociação de conteúdo pode criar variações em como os recursos são representados, e o sistema de cache precisa ser capaz de lidar com essas variações para entregar o conteúdo apropriado aos diferentes clientes. Para essa gestão ser possível é incluido nas mensagens HTTP o header Vary para indicar quais foram os headers que foram usados na negociação do conteúdo, o uso desse header faz com que o sistema de caching seja capaz de funcionar de uma forma otimizada.

No contexto da plataforma Spring MVC, indique duas formas distintas para a definição de beans.

Com Anotações ou com configuração XML

Propósito do Webpack:
Facilita o processo de construção de aplicações web. Otimiza os recursos, a gestão de depêndencias e organização de código da aplicação web.

Forma de utilização do Webpack:
Configuração a partir de um ficheiro configuração webpack.config.js.
Instalação e execução via NPM.
Plugins e Loaders para estender as suas funcionalidades.

A interface uniforme no protocolo HTTP é crucial para alcançar uma arquitetura REST eficaz, proporcionando benefícios de simplicidade, visibilidade (Os recursos são identificados por URIs, proporcionando visibilidade e identificação claras dos recursos), desacoplamento (A interface uniforme promove o desacoplamento entre clientes e servidores, ou seja, mudanças no servidor não devem exigir alterações significativas nos clientes, desde que a interface permaneça consistente) e facilidade de evolução do sistema.

A utilização de **métodos idempotentes** no desenho de APIs HTTP oferece vantagens em termos de desempenho e segurança uma vez que, os resultados destes métodos podem ser armazenados em cache, visto que, o resultado não é alterado quando o método é repetido. Além disso, estes métodos podem ser repetidos em caso de falha de comunicação, sem que o resultado seja alterado, o que permite que o cliente possa repetir o pedido sem ter de se preocupar com o estado do servidor ou a falha presente no pedido anterior.

Apesar dos browsers modernos oferecerem suporte nativo aos módulos ECMAScript, o webpack continua a ser uma ferramenta relevante para o desenvolvimento de aplicações web uma vez que, o webpack oferece uma forma simples de organizar, gerir e otimizar os recursos da aplicação web, das dependências entre esses recursos e do processo de construção da aplicação web e também uma funcionalidade chamada de "minificação" para não expor o código fonte ao utilizador, sendo esta a funcionalidade muito relevante para o desenvolvimento de aplicações web.

window.location.pathname realiza um pedido HTTP de método **GET** para o path especificado enquanto que **history.pushState** apenas modifica o URL

A utilização de [] como segundo argumento da função **useEffect**, significa que o **useEffect** vai ser apenas chamado uma vez durante o tempo de vida da instância do componente.

No protocolo HTTP, uma mensagem de resposta com status code 401 deve conter o header de resposta WWW-Authenticate.

No contexto da plataforma Spring MVC vai existir uma instância de **HttpServletRequest** para cada pedido

O propósito principal do sistema NPM é facilitar a gestão, instalação e distribuição de pacotes de software (bibliotecas, frameworks e utils) que podem ser usados no desenvolvimento de projetos. O sistema NPM também é um essencial para o controlo de versões de pacotes de software e gestão de dependências entre pacotes.

O passo de construção em desenvolvimento de aplicações para execução em browser é crucial para garantir que o código seja otimizado, compatível e seguro antes de ser entregue ao navegador. Este passo garante então a compatibilidade, segurança e desempenho em diversos navegadores e também uma melhor gerência de dependências e que recursos devem ou não ser carregados.

Indique o que é necessário realizar para que uma aplicação single page application suporte deep-linking

Seria necessário utilizar, por exemplo, um React Router para ser possível configurar as rotas da aplicação e manipular a navegação entre as páginas. Seria também necessário criar componentes capazes de atualizar o seu conteúdo automaticamente sem haver a necessidade de recarregar a página.

No contexto de uma aplicação que usa as bibliotecas React e React Router, a principal diferença entre um componente **<Link />** e um componente **<a />** é que o componente **<Link />** é capaz de atualizar o conteúdo da página sem a necessidade de recarregar a página a partir da manipulação da navegação, o mesmo não é possível com o componente **<a />** que apenas consegue navegar entre páginas recarregando a página inteira quando o link é clicado.

O propósito principal do **preventDefault()** é impedir que a ação padrão associada a um evento ocorra. Isso é útil em SPAs, onde as transições de página muitas vezes são gerenciadas de forma dinâmica no lado do cliente, sem a necessidade de recarregar a página inteira.

A link relation self deve ser usada sempre que se pretender fornecer ao cliente informações sobre um recurso sem a necessidade de o cliente ter de fazer um pedido HTTP para obter essas informações. O uso desta relação permite que o cliente possa identificar, acessar e manipular o recurso em questão, sem depender de pedidos adicionais, proporcionando uma forma mais eficiente de interação com o recurso.

No contexto da framework React, a utilização de **undefined** como segundo argumento da função **useEffect**, significa que o efeito vai ser chamado sempre que a função que define o componente for chamada.

Na plataforma **browser** o resultado da avaliação de “const m = require(‘utils’)” num módulo da aplicação resulta num Erro de execução, porque a função **require** não está definida

Na platforma Spring MVC e assumindo a configuração por omissão: Os handlers presentes numa classe controller podem ser chamados em concorrência sobre a mesma instância, por threads distintas.

GET: Idempotente e safe
POST: Não idempotente e não safe
PUT: Idempotente e não safe
HEAD: Idempotente e safe
DELETE: Idempotente e não safe

A utilização de módulos **NPM** usando o sistema **CommonJS** em ficheiros Javascript destinados à execução em browsers requer a utilização da ferramenta webpack ou similar

No protocolo HTTP, o conceito de interface uniforme significa que o significado do status codes nas mensagens de resposta não depende do recurso acedido

No protocolo HTTP, o conceito de interface uniforme significa que a semântica de um header de resposta não depende do recurso alvo e que o significado do status codes nas mensagens de resposta não depende do recurso acedido.

Para se suportar deep-linking no contexto de uma single page application é necessário configurar o servidor que serve essa aplicação da seguinte forma: Caso o caminho presente num pedido GET não esteja associado a um ficheiro, então é retornada uma resposta de sucesso com o conteúdo de index.html em vez de uma resposta com status code 404.

No contexto da utilização da biblioteca Spring MVC, a execução da função doFilter pertencente à interface **HttpFilter** ocorre sempre no contexto da thread associada ao pedido HTTP que resultou nesta chamada.

Siren:
-Uma especificação hypermedia para representar entidades de maneira fácil para navegação ou ações relacionadas.
-Inclui representação de ações em representação de recursos.
-Agrupando as propriedades de recursos non-linked em **properties**.
-A propriedade **links** é usada para representar links navegacionais para outros recursos, páginas incluem sempre um link referência para si mesmos.
-O atributo **rel** contém o tipo de relação entre dois recursos, eventualmente representado como link.
-Siren é media type: **application/vnd.siren+json**.

```
@RestController
class Controller (private val storage: Storage) {
  @GetMapping("/failures")
  fun getFailures(): ResponseEntity<OutputModel> {
    val rsp = FailuresOutputModel(storage.get())
    return ResponseEntity.status(200).contentType(MediaType.APPLICATION_JSON).body(rsp)
  }
}
```

No contexto da biblioteca React, uma função **hook** pode ser chamada dentro de outra função **hook**

```
@RestController
class Controller {
  @GetMapping("/debug")
  fun debug(response: HttpServletResponse) {
    ResponseEntity.ok("Hello Debug")
  }
}

// Não ter um return no Controller torna possível a adição do header na response
```

```
@Component
class Interceptor(private val storage: Storage) : HandlerInterceptor {
  override fun postHandle(
    request: HttpServletRequest,
    response: HttpServletResponse,
    handler: Any,
    modelAndView: ModelAndView?
  ) {
    if (response.status < 500 || request.requestURI == "/failures") return
    val handlerMethod = handler as HandlerMethod
    val failure = Failure(
      method = request.method,
      uri = request.requestURI,
      status = response.status,
      controller = handlerMethod.beanType.simpleName,
      handler = handlerMethod.method.name
    )
    storage.add(failure)
  }
}
```

```
@Component
class Storage {
  private val lock = ReentrantLock()
  private val map = mutableListOf<Failure>()
  fun get(): List<Failure> {
    lock.withLock {
      return map.takeLast(10)
    }
  }
  fun add(failure: Failure) {
    lock.withLock {
      map.add(failure)
    }
  }
}
```

```
private val logger = LoggerFactory.getLogger(<className>::class.java)
logger.info("Conteúdo")
```

```
@Component
class Interceptor: HandlerInterceptor {
  override fun preHandle(request: HttpServletRequest, response: HttpServletResponse, handler: Any): Boolean {
    val handlerMethod = handler as HandlerMethod
    request.setAttribute("method", handlerMethod.method.name)
    request.setAttribute("startTime", System.currentTimeMillis())
    return true
  }
  override fun postHandle(
    request: HttpServletRequest,
    response: HttpServletResponse,
    handler: Any,
    modelAndView: ModelAndView?
  ) {
    val method = request.getAttribute("method") as String
    val startTime = request.getAttribute("startTime") as Long
    val endTime = System.currentTimeMillis()
    val duration = endTime - startTime
    val header = "$method took $duration ms"
    response.addHeader("HTTP Debug", header)
  }
}
```

```
if (request.requestURI.startsWith("/status"))

if (request.cookies == null ||
request.getHeader("Authorization") == null)
```

```
useEffect(() => {
  setPending(true)
  awaitPromise()
  async function awaitPromise() {
    try {
      const result = await props.f()
      setResult(result)
    } catch (e) {
      setResult(e.message)
    } finally {
      setPending(false)
    }
  }
}, [props.f])
```

```
async function doFetch(uri: string) {
    try {
      const startTime = Date.now()
      const response = await fetch(uri, {method: "GET"})
      const text = await response.text()
        setStatus(response.status)
        setBody(text)
      }
    } catch (e) {
      setStatus(e.status})
      setBody(e.message)
    }
  }
```

```
type props = {
  uri: string,
  period: number
}
```

```
type props = {
  f: () => Promise<string>
}
```

```
return (
    <div>
      <ul>
        {list.map((item, index) => (
          <li key={index}>{item.url},{item.text}</li>
        ))}
      </ul>
    </div>
  )
```

```
async function doFetch() {
  for (const currItem of props.list) {
    setList((x) => [...x, {url: currItem, text: "..."}])
    try {
      const fetched = await fetch(currItem)
      setList((x) => {
        const newList = [...x]
        newList[x.length - 1].text = fetched.status.toString()
        return newList
      })
    } catch (e) {
      setList((x) => {
        const newList = [...x]
        newList[x.length - 1].text = e.message
        return newList
      })
    }
  }
}
```

```
function useInput(initial: string): [
currentValue: string,
changeHandler: (ev: React.ChangeEvent<HTMLInputElement>) => void
] {
  const [value, setValue] = useState<string>(initial)

  const changeValue =
(ev: React.ChangeEvent<HTMLInputElement>) => {
    setValue(ev.target.value)
  }

  return [value, changeValue]
}
```

```
function useCounter(initial: number):
[observed: number, inc: () => void, dec: () => void] {
  const [count, setCount] = useState(initial)
  const inc = () => setCount((x) => x + 1)
  const dec = () => setCount((x) => x - 1)
  return [count, inc, dec]
}
```

```
useEffect(() => {
    if (pending) {
      const tid = setInterval(() => {
        setCounter((x) => x + 1)
      }, 100)
      return () => clearInterval(tid)
    }
  }, [pending])
```