Aplicações Web com Java Baseadas em Spring

- Spring
  - Framework para construção de aplicações
  - Baseado no conceito de inversão de controle (Inversion of Control) e na Injeção de Dependência (Dependency Injection)
    - Inversion of Control (IoC)
      - O fluxo é baseado nos objetos criados e não o contrário
    - Dependendecy Injection (DI)
      - Um objeto recebe os objetos que ele depende (dependências) assim o comportamento pode ser "injetado" em outros objetos
  - Atualmente é o framework mais popular na linguagem Java
- Composto por vários módulos divididos em projetos (https://spring.io/projects)
  - Spring Framework
    - Suporte básico para injeção de dependência, gestão de transações, web apps, acesso a dados e mensagens
  - Spring Boot
    - Permite a criação rápida de aplicações baseadas no Framework Spring
  - Spring Data
    - Acesso a dados em banco de dados relacionais e não relacionais
  - Spring Cloud, Spring Cloud Data Flow, Spring Security, Spring Session, etc.
- https://docs.spring.io/spring-framework/docs/current/reference/html/

Dependendecy Injection (DI) Exemplo

- Sem DI

```
class Car{
  private Wheel wh = new NepaliRubberWheel();
  private Battery bt = new ExcideBattery();
  //…
}
```

- Com DI

```
class Car{
  private Wheel wh; // Inject an Instance of Wheel (dependency of car) at runtime
  private Battery bt; // Inject an Instance of Battery (dependency of car) at runtime

 //setter Injection
  void setWheel(Wheel wh) {
      this.wh = wh;
  }
}
```

Inversion of Control (IoC) Exemplo

- Sem IoC

```
public class TextEditor {
    private SpellChecker checker;
    public TextEditor() {
        this.checker = new SpellChecker();
    }
}
```

- Com IoC

```
public class TextEditor {
    private IocSpellChecker checker;
    public TextEditor(IocSpellChecker checker) {
        this.checker = checker;
    }
}
SpellChecker sc = new SpellChecker(); // dependency
TextEditor textEditor = new TextEditor(sc);
```

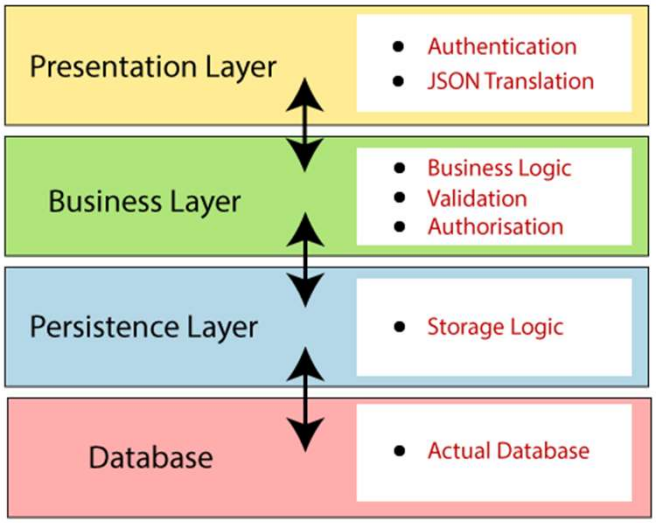
- A classe TextEditor possui o controle sobre o objeto da classe SpellChecker

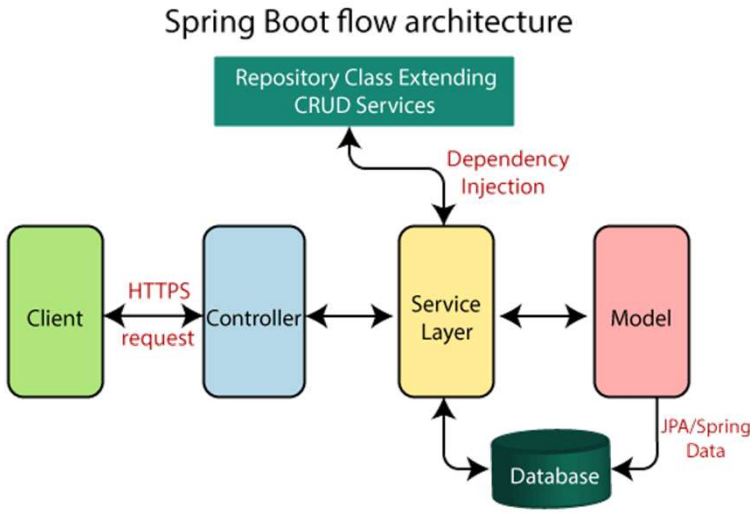
Spring Boot

- Utiliza o conceito de Rapid Application Development (RAD)
- Permite a criação de aplicações java “stand-alone” com uma configuração mínima do Spring (execução utilizando apenas java –jar)
- Remove a necessidade de configuração de arquivos XML (como o “Deployment Descriptor”) simplificando o uso do Spring
  - Spring MVC necessita de diversas configurações manuais
- Características
  - Utiliza a abordagem de Dependency Injection do Spring
  - Contém suporte a diversos banco de dados
  - Simplifica a integração com outros frameworks Java (como JPA/Hibernate ORM)
  - Reduz o tempo de desenvolvimento de uma aplicação
- Uma desvantagem é o número de dependências que são adicionadas ao projeto
- Suporte em diversos IDE: Eclipse, IntelliJ IDEA Ultimate, NetBeans, Sprint Tools Suite (STS)
- Possui ambiente próprio para desenvolvimento (Spring Tools)
  - Spring Tools 4 - https://spring.io/tools
- Documentação atual do Spring Boot
  - https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

Spring Boot Arquitetura

- Camadas
  - Apresentação
    - Recebe as requisições HTTP
  - Persistência
    - Spring Boot cria todas as classes necessárias para
- Fluxo de Informação
  - Cliente: Navegador ou aplicação
    - HTTP Request pode receber como resposta JSON
  - Controller
    - Recebe as requisições e através e faz o mapeamento para lógica de serviço (Business Logic)
  - Camada de Serviço
    - Executa a lógica necessária para o serviço
    - Executa operações sobre os dados do Model que estão mapeados no Banco de dados utilizando JPA





Spring Annotations

- O Spring utiliza diversas anotações que facilitam o processo de configuração e escrita do código das aplicações
- A anotação é um metadado que é associado à uma classe, método ou atributo
- A Plataforma java possui um conjunto padrão de anotações e permite a sua extensão
- O Spring faz um uso intenso de anotações
- O uso de anotações libera da necessidade de ter diversos arquivos de configuração indicando como cada parte do Código deve ter tratada

Spring Application

```
package com.example.hellospringboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloSpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloSpringBootApplication.class, args);
    }
}
```

- @SpringBootApplication
  - Anotação básica que indica o ponto de entrada de uma aplicação baseada no Spring
  - É uma combinação de três anotações
    - @Configuration – indica que trata-se de uma classe de configuração do Spring
    - @ComponentScan – permite que o *controller* busque pelos componentes no código o pacote da aplicação (java package), facilitando a configuração
    - @EnableAutoConfiguration – permite a autoconfiguração e registro dos componentes existentes

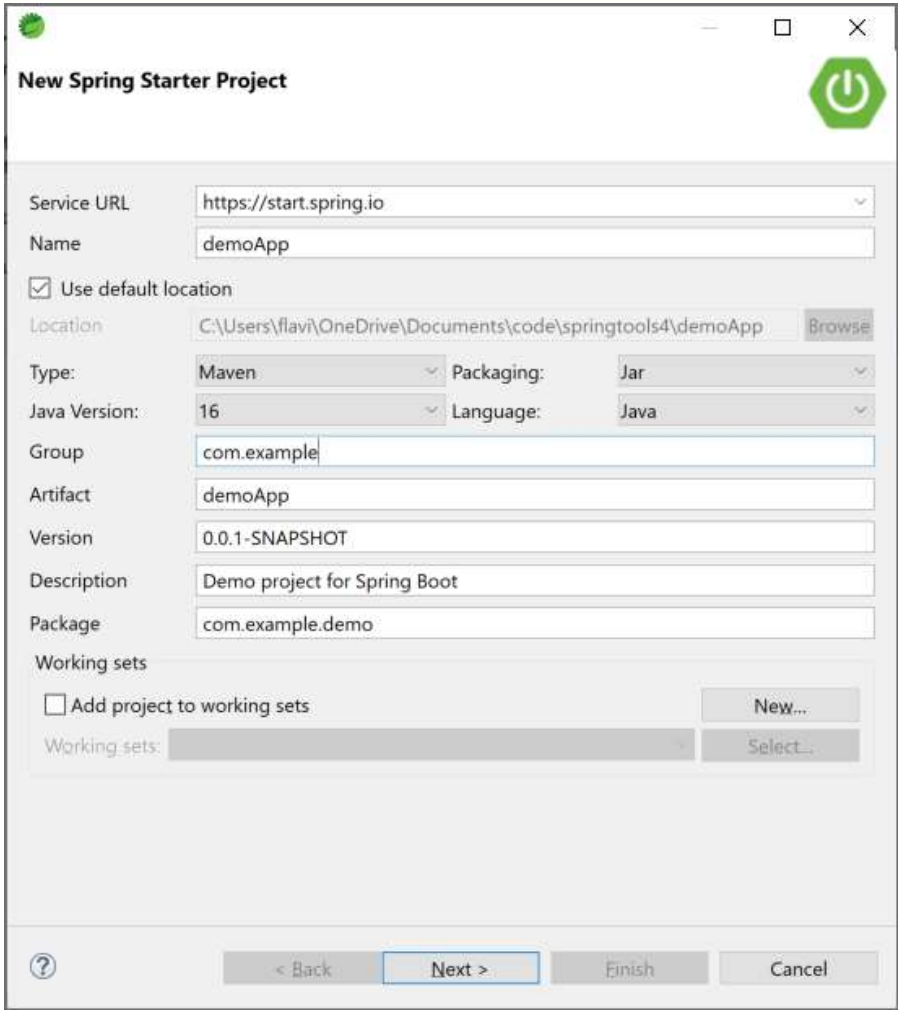
Spring Controller

```
package com.example.hellospringboot;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class RestAPI {
    @GetMapping(value = "/")
    public String getMethodName(){
        return "Hello!";
    }
}
```

- RestController é a classe responsável por tratar as requisições HTTP
- GetMapping
  - A classe pode conter vários métodos e para cada método há um mapeamento entre a URL da aplicação e o método da classe java

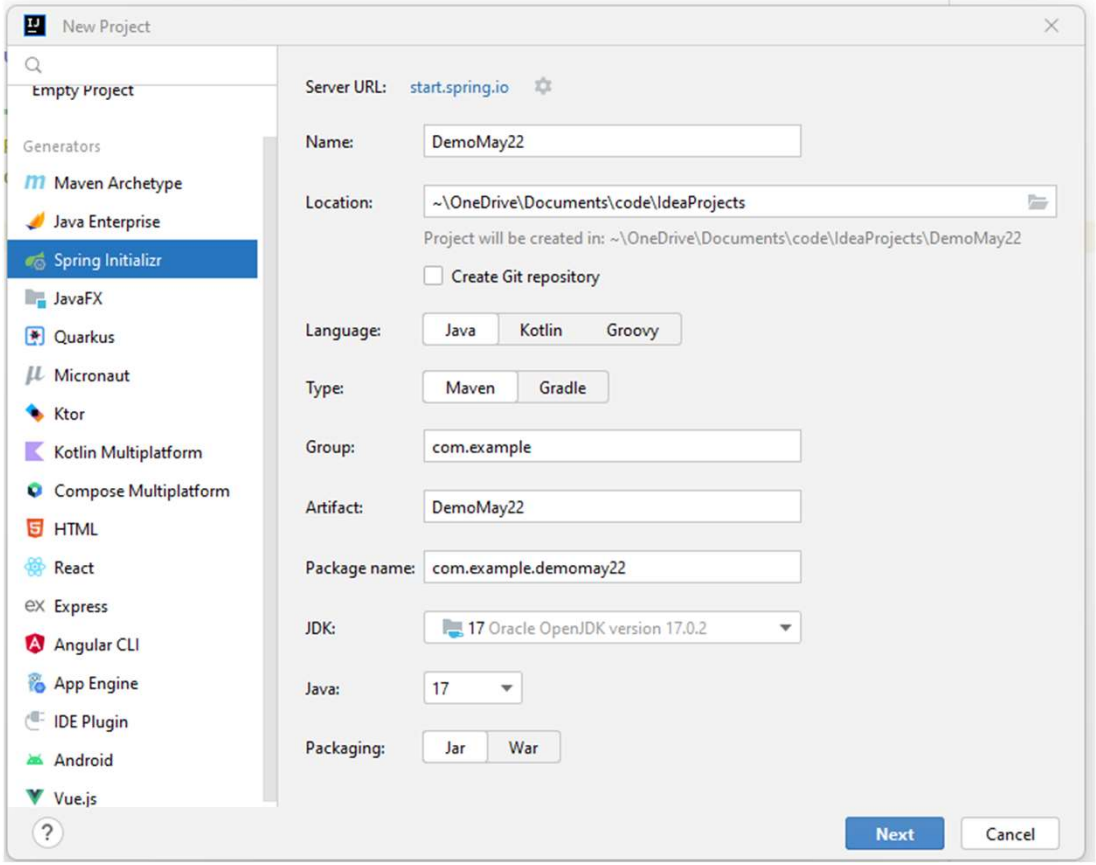
Utilizando o Spring Starter Project Página Inicial

- Permite criar um projeto compatível com Spring incluindo todas as dependências
- Pode ser acessado diretamente pela web: https://start.spring.io/
  - Neste caso é necessário o download de um ZIP do projeto.
- Name: Nome da Aplicação
- Location: Pasta onde o projeto será criado localmente
- Package: nome do pacote onde as classes java serão criadas
- Type
  - Define a forma que a aplicação será criada (build) e como as dependências serão tratadas
  - MAVEN
  - GRADLE
- Java Version
- Packaging
  - Contém código java e outros recursos da aplicação (Imagens, JS, CSS, etc.
  - Jar - Java ARchive. Aplicação para linha de comando
  - War - Web application ARchive. Aplicação para contêiner de aplicações web



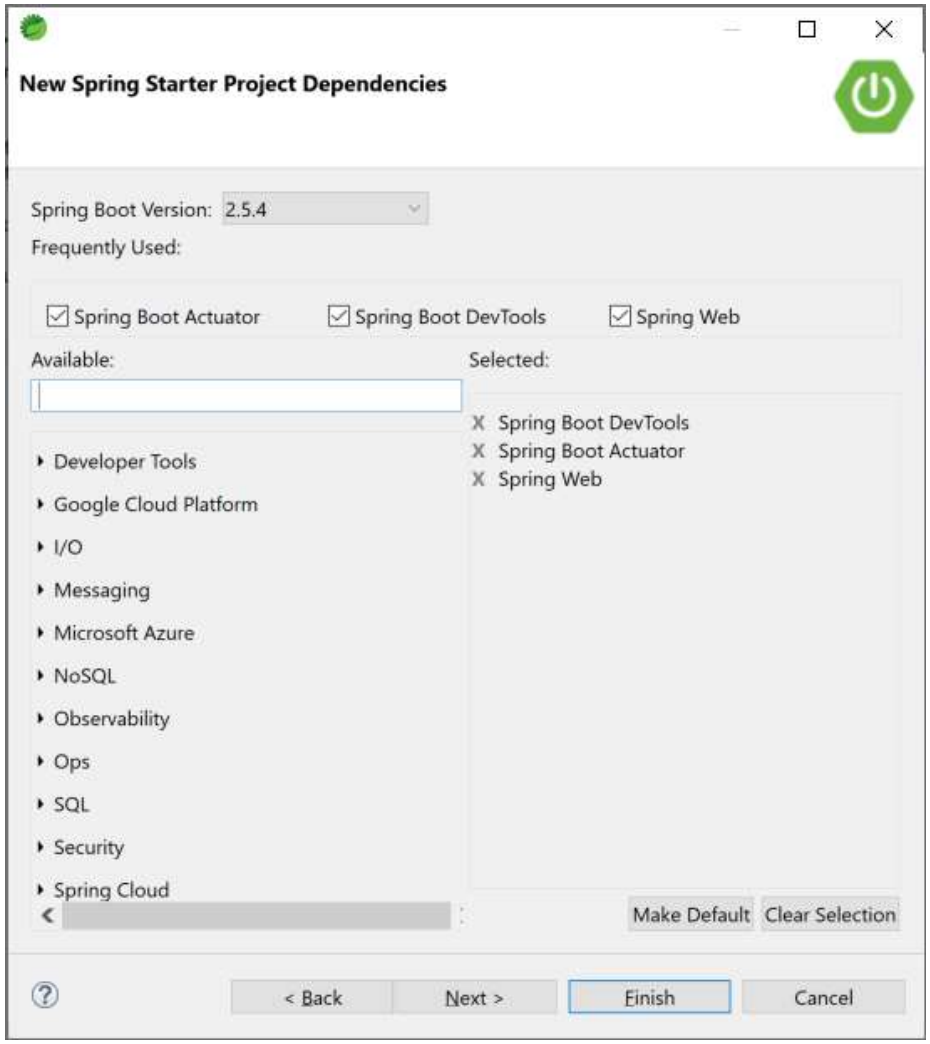
Utilizando o Spring Starter Project Página Inicial

- Permite criar um projeto compatível com Spring incluindo todas as dependências
- Pode ser acessado diretamente pela web: https://start.spring.io/
  - Neste caso é necessário o download de um ZIP do projeto.
- Name: Nome da Aplicação
- Location: Pasta onde o projeto será criado localmente
- Package: nome do pacote onde as classes java serão criadas
- Type
  - Define a forma que a aplicação será criada (build) e como as dependências serão tratadas
  - MAVEN
  - GRADLE
- Java Version
- Packaging
  - Contém código java e outros recursos da aplicação (Imagens, JS, CSS, etc.
  - Jar - Java ARchive. Aplicação para linha de comando
  - War - Web application ARchive. Aplicação para contêiner de aplicações web



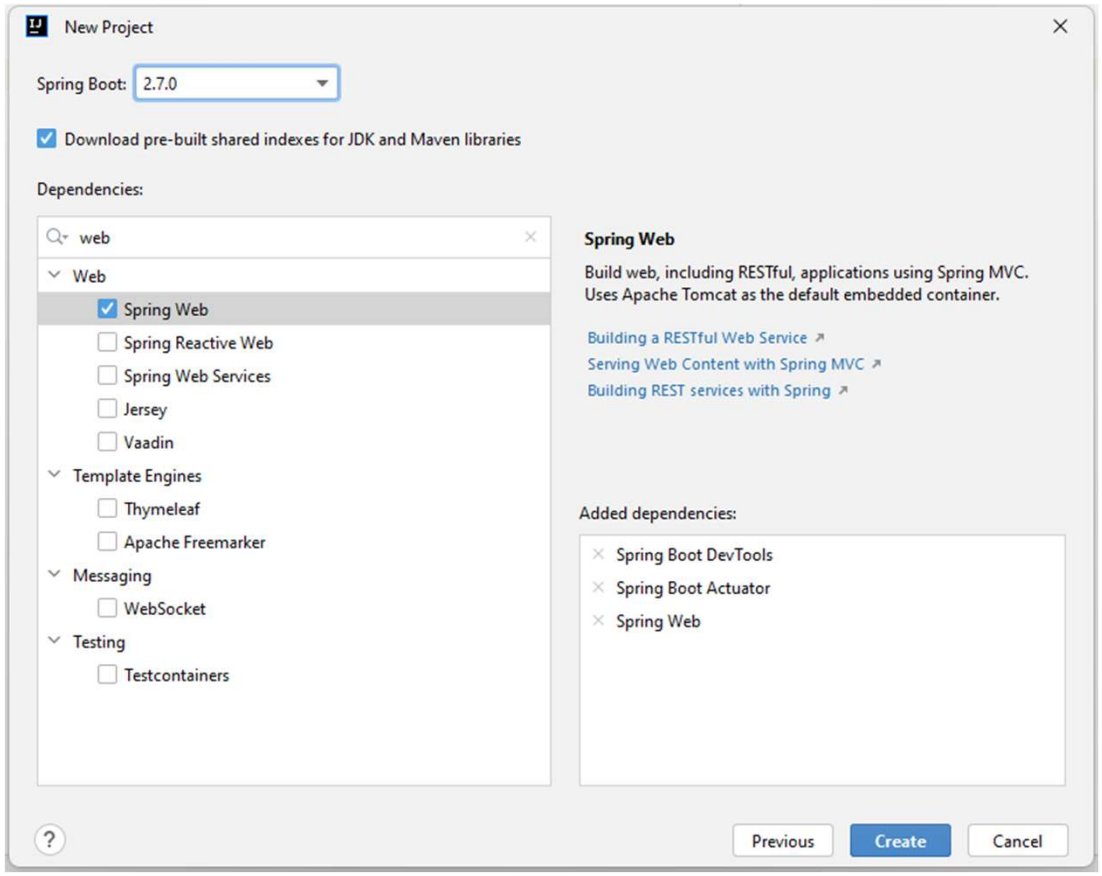
Utilizando o Spring Starter Project Dependências

- O Spring possui diversos components
- Permite selecionar os components
- Spring Boot DevTools
  - Permite reiniciar rapidamente a aplicação após modificações no Código-fonte e configurações e LiveReoad
  - https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools
- Spring Web
  - Permite construir aplicações web, baseadas no conceito RESTFul utilizando o Spring MVC
  - O Apache TOMCAT é o container padrão utilizado
  - https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#spring-web
- Spring Actuator
  - Oferece endpoints para monitorar e gerenciar a aplicação
  - https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html
- Outros projetos: https://spring.io/projects



Utilizando o Spring Starter Project Dependências

- O Spring possui diversos components
- Permite selecionar os components
- Spring Boot DevTools
  - Permite reiniciar rapidamente a aplicação após modificações no Código-fonte e configurações e LiveReoad
  - https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools
- Spring Web
  - Permite construir aplicações web, baseadas no conceito RESTFul utilizando o Spring MVC
  - O Apache TOMCAT é o container padrão utilizado
  - https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#spring-web
- Spring Actuator
  - Oferece endpoints para monitorar e gerenciar a aplicação
  - https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html
- Outros projetos: https://spring.io/projects



Recuperando Parâmetros

```
package com.example.hellospringboot;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class RestAPI {
    @GetMapping(value="/hello")
    public String sayHello(@RequestParam(value = "myName", defaultValue = "World") String
name) {
        return String.format("Hello %s!", name);
    }
}
```

- @RequestParam
  - Anotação que permite extrair parâmetros diversos de um pedido (HttpRequest)
  - No exemplo acima o parâmetro com valor “myName” é obtido e colocado na String

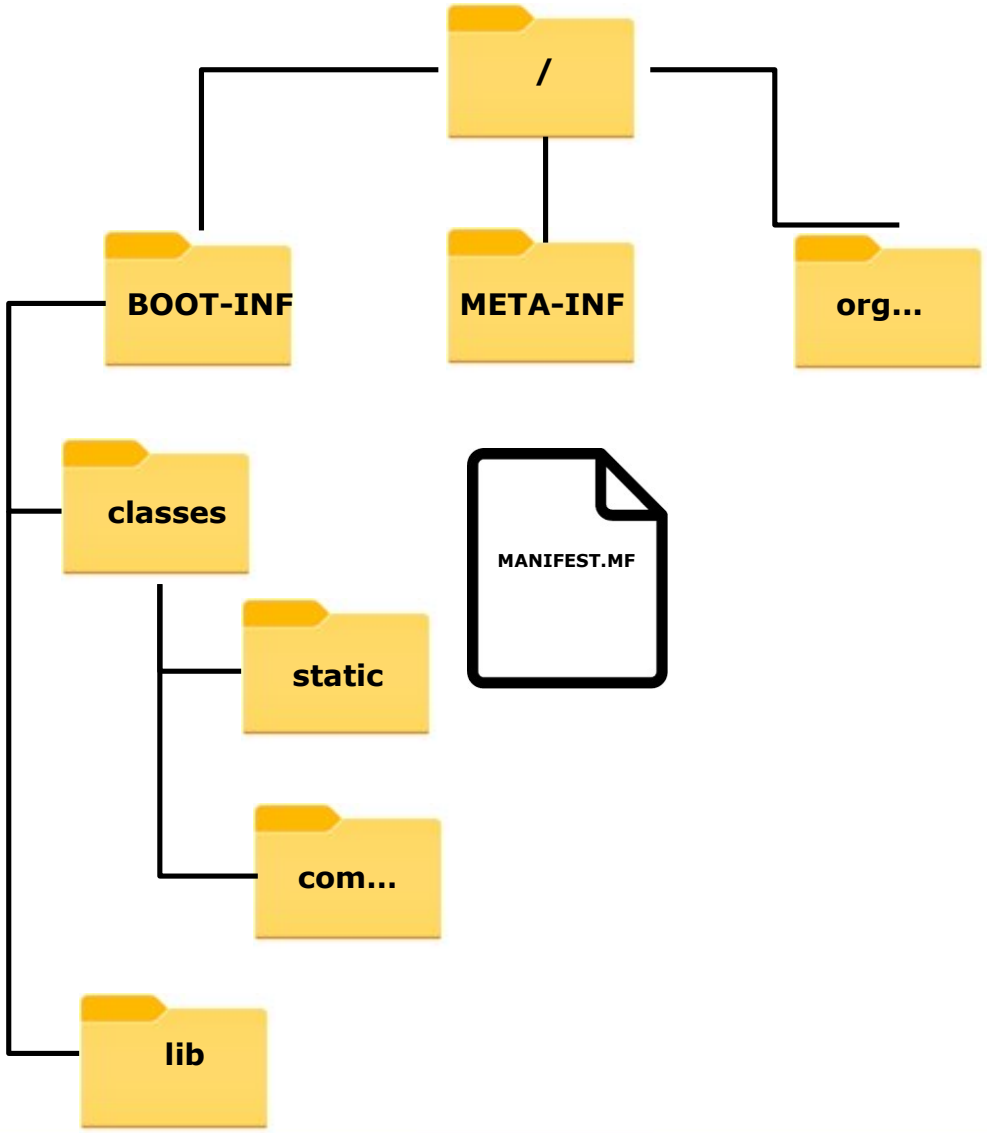
Enviando Parâmetros para o Servidor

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Your first Spring application</title>
</head>
<body>
<p><a href="/hello">Greet the world!</a></p>
<form action="/hello" method="GET" id="nameForm">
  <div>
    <label for="nameField">How should the app call you?</label>
    <input name="myName" id="nameField">
    <button>Greet me!</button>
  </div>
</form>
</body>
</html>
```

- Ligando o formulário com o Serviço
  - No formulário acima, a caixa de texto (<input>) possui o nome “myName”
  - A propriedade ação (action) do formulário aponta para a “/hello”
  - A enviar o formulário o spring faz o mapeamento entre “/hello” e o método
    - public String sayHello(String)

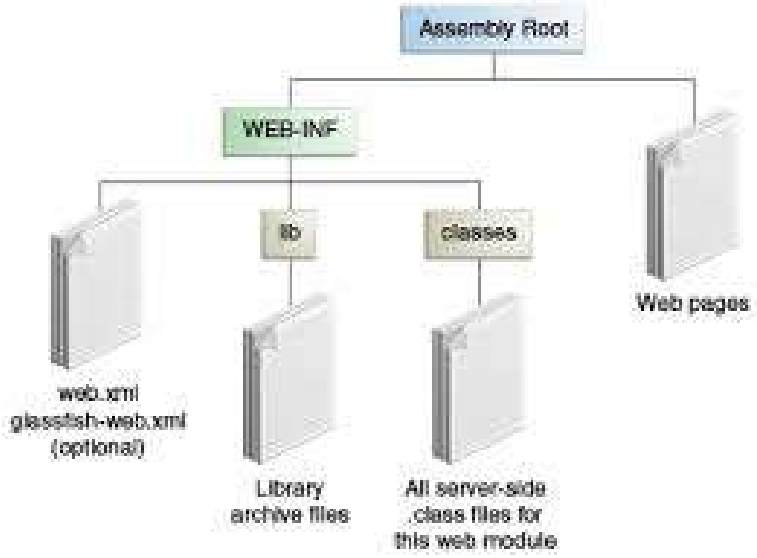
JAR FILE
Spring Boot

- / (Raiz da Aplicação)
- /META-INF
  - Contém o arquivo MANIFEST.MF
- /<PACKAGENAME>/classes
  - Todas as classes java compiladas (.class) do Spring Boot são colocadas neste local
  - Segue a estrutura de pacotes padrão do Spring: /org/springframework/boot/loader
- /BOOT-INF
  - /classes
    - No caso de aplicações baseadas no Spring Boot esta pasta contém todas classes java compiladas (.class) seguindo a estrutura de pacotes definida
    - /static
      - Contém *resources* estáticos (como arquivos de propriedades)
  - /lib
    - Contém todas as bibliotecas e dependências utilizadas pela aplicação criada com os diversos componentes do framework Spring (*.jar)



WAR File

- / (Raiz da Aplicação)
  - Todos os arquivos de acesso públicos são colocados nesta pasta como por exemplo: HTML; JSP; GIF; etc.
  - No caso do TOMCAT o diretório raiz da aplicação é criado dentro da pasta **webapps**
- /WEB-INF
  - Os arquivos desta pasta não são de acesso público
  - Esta pasta contém um arquivo chamado **web.xml, conhecido como** deployment descriptor, que contém as configurações para uma aplicação WEB
- /WEB-INF/classes
  - Todas as classes java compiladas (.class) são colocadas neste local
- /WEB-INF/lib
  - Todas as classes que estão compactadas em um arquivo JAR são colocadas neste diretório

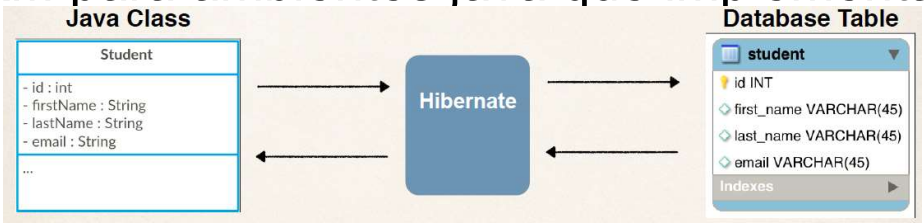


Spring Data
Persistência de Dados

- O Spring Data é um projeto do framework Spring
  - https://spring.io/projects/spring-data
  - https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/
- Permite a abstração do acesso aos mais diferentes tipos de persistência de dados utilizando os conceitos do Spring
- Fornece acesso à diversos tipos de persistência tanto relacionais quanto não-relacionais e ainda à serviços em nuvem de acesso à dados
- Suporte (entre outros)
  - Java Database Connectivity (JDBC) - https://spring.io/projects/spring-data-jdbc
  - Java Persistence API (JPA) - https://spring.io/projects/spring-data-jpa
    - Facilita a integração com bases que suporta JPA
  - KeyValue - https://github.com/spring-projects/spring-data-keyvalue
    - Acesso a banco de dados não relacionais, frameworks de map-reduce e serviços de dados em cloud
  - LDAP - https://spring.io/projects/spring-data-ldap
    - Suporte à repositórios baseados em Lightweight Directory Access Protocol (LDAP)
  - MongoDB - https://spring.io/projects/spring-data-mongodb
    - Integração com MongoDB, um banco baseado em documento (NoSQL)
  - Redis - https://spring.io/projects/spring-data-redis
    - Suporte ao Remote Dictionary Server (Redis); banco de dados em memória baseado em key-value
  - REST - https://spring.io/projects/spring-data-rest
    - Permite expor repositórios utilizando o conceito de REST
  - Apache Cassandra - https://spring.io/projects/spring-data-cassandra
    - Suporte banco de dados NoSQL Cassandra
  - Apace Geode
    - Suporte ao Apache Geode; Banco de Dados baseado em Memória NVRAM

Java Persistence API (JPA) Object-Relational Mapping (ORM)

- O JPA é uma API para mapear Objetos em Tabelas Relacionais
- Há uma incongruência entre o conceito de objetos (todo/parte) com tabelas relacionais
  - Necessário mapear o conceito de objetos em tabelas
- O Hibernate é um framework para ORM para ambientes java que implementa a JPA.



- O Hibernate é um JPA provider
- O Spring Data JPA simplifica o processo de uso do JPA com uma camada adicional de abstração do acesso à dados
- O Spring Data JPA pode ser utilizado com qualquer provider da JPA
  - O Hibernate, como implementação de referência do JPA, pode ser este provider

Entidade

- Conceito do Model (Modelo do Domínio) mapeado em uma ou mais tabelas
  - São coisas, conceitos, papéis ligados à uma área de conhecimento
  - Exemplos: Produto, Venda, Cliente, etc.
- Anotações básicas da JPA permitem associar Classe com Tabela
  - @Entity
    - Associada nome da classe com tabela no banco de dados
  - @Id
    - Utilizado para indicar qual é a chave primária associada ao objeto
  - @GeneratedValue
    - Indica como o valor do atributo será Gerado
- Lista completa das anotações do JPA utilizadas pelo Spring
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference

Exemplo de Entidade

```
package com.example.accessingdatajpa;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Customer {
                @Id
                @GeneratedValue(strategy=GenerationType.AUTO)
                private Long id;
                private String firstName;
                private String lastName;
                protected Customer() { }
                public Customer(String firstName, String lastName) {
                  this.firstName = firstName;
                  this.lastName = lastName;
                }
                @Override
                public String toString() {
                  return String.format(
                  "Customer[id=%d, firstName='%s', lastName='%s']",
                  id, firstName, lastName);
                }
                //getters
}
```

Operações CRUD CrudRepository

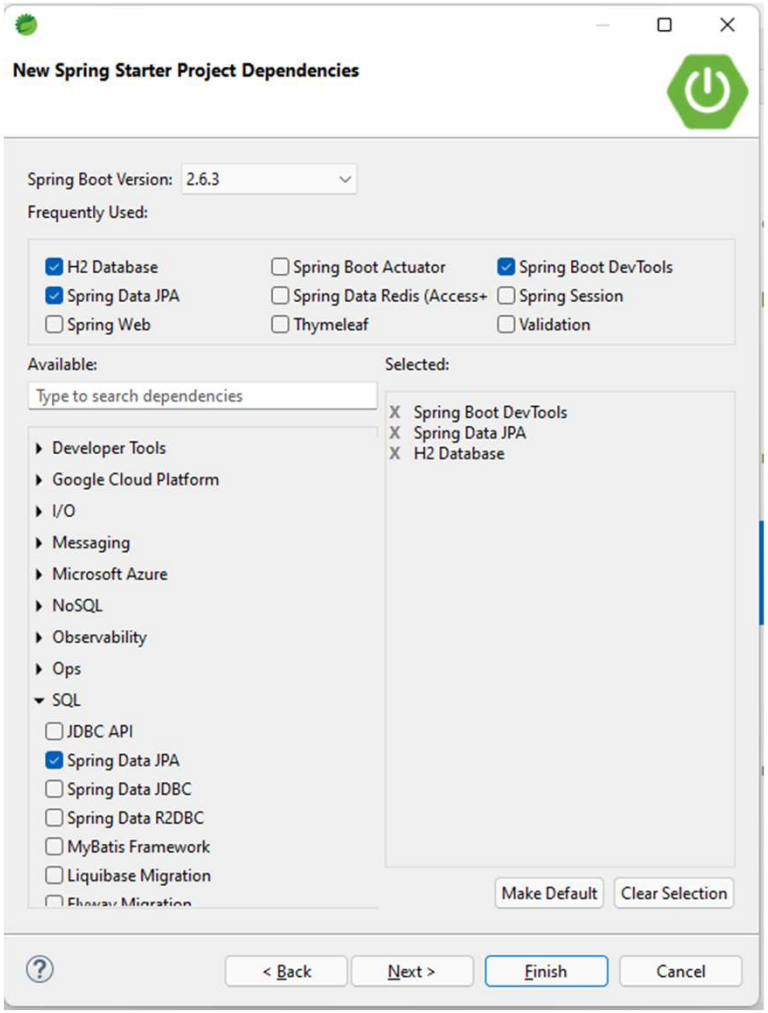
- CrudRepository é uma interface Java oferecida pelo Spring
- Permite operações de Create, Read, Update, Delete (CRUD) em objetos java
  - SQL
    - Create = Insert into...
    - Read = Select * from
    - Update = update
    - Delete = delete
  - https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html
- Spring Data necessita apenas da definição de uma interface
  - Interface precisa estender CrudRepository
  - Interface define os métodos que serão expostos e os tipos de objetos (entidades) envolvidos em cada método

Operações CRUD
CrudRepository - Exemplo

```
package com.example.accessingdatajpa;
import java.util.List;
import org.springframework.data.repository.CrudRepository;
public interface CustomerRepository extends CrudRepository<Customer, Long> {
  List<Customer> findByLastName(String lastName);
  Customer findById(long id);
}
```

Spring Starter
Aplicação básica utilizando JPA

- Componentes Necessários
  - Spring Boot Dev Tools
    - Inclui algumas funcionalidades para facilitar o desenvolvimento de aplicações, como:
    - "LiveReload" - recarrega os arquivos para o navegador quando arquivos estáticos (html, CSS, JS) são alterados
    - Reinicio automático quando o código Java é alterado
  - Spring Data JPA
    - Suporte à persistência utilizando JPA
  - H2 Database
    - Suporte para o banco de dados H2
    - Para o H2 o Spring faz uma configuração utilizando o H2 que está embarcado como uma dependência
    - Neste caso é necessário nenhuma configuração adicional
    - Maiores informações:
      - http://www.h2database.com/html/main.html



Aplicação Exemplo 1/3

```
package com.example.accessingdatajpa;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
@SpringBootApplication
public class AccessingDataJpaApplication {

  private static final Logger log = LoggerFactory.getLogger(AccessingDataJpaApplication.class);

  public static void main(String[] args) {
    SpringApplication.run(AccessingDataJpaApplication.class);
  }
```

Aplicação Exemplo 2/3

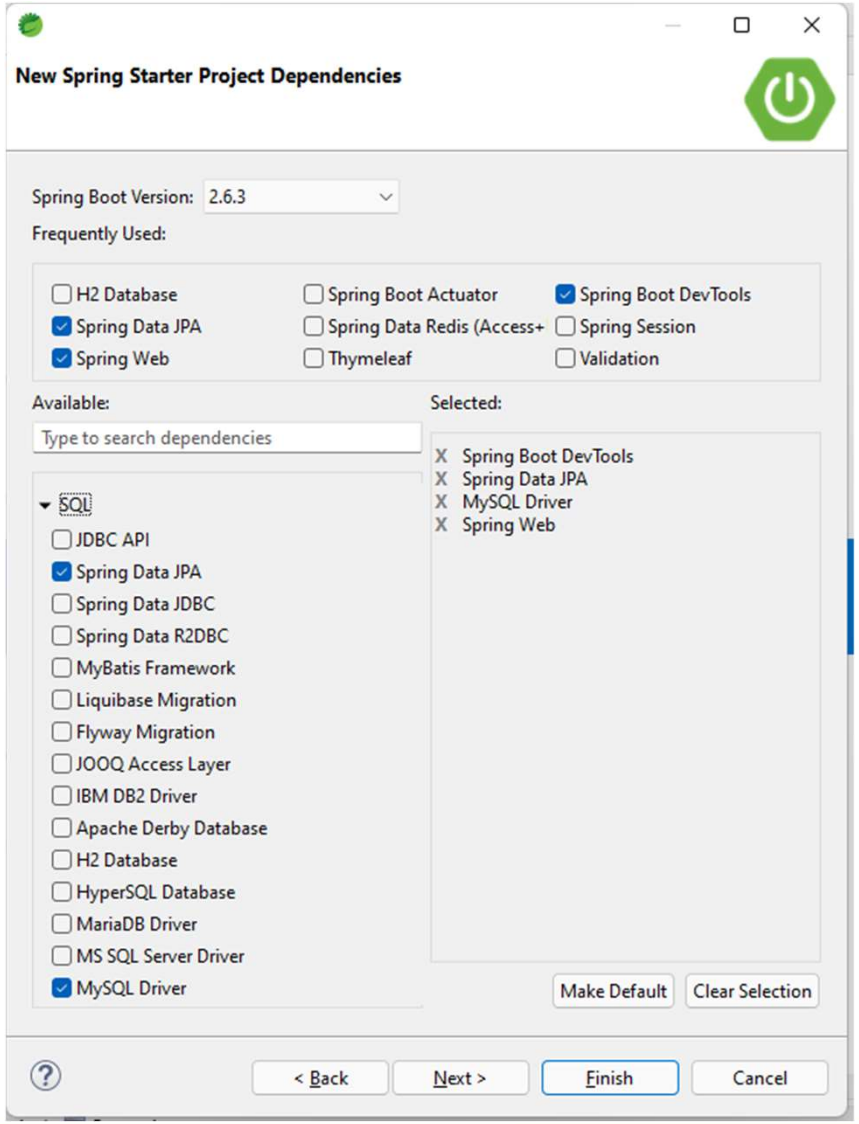
```
@Bean
public CommandLineRunner demo(CustomerRepository repository) {
  return (args) -> {
    // save a few customers
    repository.save(new Customer("Jack", "Bauer"));
    repository.save(new Customer("Chloe", "O'Brian"));
    repository.save(new Customer("Kim", "Bauer"));
    repository.save(new Customer("David", "Palmer"));
    repository.save(new Customer("Michelle", "Dessler"));
    // fetch all customers
    log.info("Customers found with findAll():");
    log.info("-------------------------------");
    for (Customer customer : repository.findAll()) {
      log.info(customer.toString());
    }
    log.info("");
```

Aplicação Exemplo 3/3

```
// fetch an individual customer by ID
      Customer customer = repository.findById(1L);
      log.info("Customer found with findById(1L):");
      log.info("--------------------------------");
      log.info(customer.toString());
      log.info("");

      // fetch customers by last name
      log.info("Customer found with findByLastName('Bauer'):");
      log.info("--------------------------------------------");
      repository.findByLastName("Bauer").forEach(bauer -> {
        log.info(bauer.toString());
      });
      // for (Customer bauer : repository.findByLastName("Bauer")) {
      //  log.info(bauer.toString());
      // }
      log.info("");
    };
  }
}
```

Spring Data MySQL

- Utilizando o Java Persistence API (JPA) é possível a conexão com diferentes banco de dados
- Exemplos
  - MySQL; Postgres; Oracle; DB2; SQLServer, etc.
- A seguir será mostrado como utilizar o MySQL
- Spring Starter
  - Spring DevTools
  - Spring Web
    - Suporte para uma aplicação WEB
    - Utiliza o Apache Tomcat como o default contêiner
  - Spring Data JPA
  - MySQL Driver
    - Adiciona o driver MySQL para a aplicação



Aplication

```
package com.example.accessingdatamysql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataMysqlApplication {

        public static void main(String[] args) {
                SpringApplication.run(AccessingDataMysqlApplication.class, args);
        }
}
```

Entidade

```
package com.example.accessingdatamysql;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity // This tells Hibernate to make a table out of this class
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  private String name;
  private String email;
  //getters
 //setters
}
```

Operações CRUD
CrudRepository - Exemplo

```
package com.example.accessingdatamysql;
import org.springframework.data.repository.CrudRepository;
import com.example.accessingdatamysql.User;
// This will be AUTO IMPLEMENTED by Spring into a Bean called
userRepository
// CRUD refers Create, Read, Update, Delete

public interface UserRepository extends CrudRepository<User, Integer> {

}
```

Controller – Exemplo 1/2

```
package com.example.accessingdatamysql;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
@Controller  // This means that this class is a Controller
@RequestMapping(path="/demo") // This means URL's start with /demo (after Application path)
public class MainController {
  @Autowired // This means to get the bean called userRepository
   // Which is auto-generated by Spring, we will use it to handle the data
   private UserRepository userRepository;
   @PostMapping(path="/add") // Map ONLY POST Requests
   public @ResponseBody String addNewUser (@RequestParam String name, @RequestParam String email) {
     // @ResponseBody means the returned String is the response, not a view name
     // @RequestParam means it is a parameter from the GET or POST request
     User n = new User();
     n.setName(name);
     n.setEmail(email);
     userRepository.save(n);
     return "Saved";
}
```

```
@GetMapping(path="/all")
```

Controller – Exemplo 2/2

```
  @GetMapping(path="/all")
   public @ResponseBody Iterable<User> getAllUsers() {
       // This returns a JSON or XML with the users
       return userRepository.findAll();
   }
}
```

Controller – Exemplo 2/2

```
  @GetMapping(path="/all")
   public @ResponseBody Iterable<User> getAllUsers() {
       // This returns a JSON or XML with the users
       return userRepository.findAll();
   }
}
```

Homepage – index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>MYSQL Add Form</title>
</head>
<body>
<p><a href="/demo/">AddUser!</a></p>
<form action="/demo/add" method="POST" id="nameForm">
  <div>
    <label for="nameField">Input Your Data</label>
    <input name="name" id="nameField">
    <input name="email" id="emailField">
    <button>Add User</button>
  </div>
</form>
</body>
</html>
```

Banco de Dados
Parâmetros de Conexão

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
spring.datasource.driver-class-name =com.mysql.jdbc.Driver
#spring.jpa.show-sql: true
```

Spring Data Repositórios

- O Spring Data oferece alguns tipos de repositórios
- CrudRepository
  - Oferece funcionalidades básicas para as operações CRUD
  - Não está ligada a nenhuma tecnologia de persistência
  - https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html
- PagingAndSortingRepository
  - Fornece métodos para paginação e ordenação
  - https://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html
- JPARepository
  - Contém todas as funcionalidades de CrudRepository e PagingAndSortingRepository
  - https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

```
@Repository
public interface EntityClassRepository extends JpaRepository<EntityClass, Long>
```

Spring Data CrudRepository

- Operações sobre entidades

1. Realiza a persistência (Create, Update)
2. Retorna uma entidade a partir do seu ID (Read)
3. Retorna todas as entidades
4. Retorna o número de entidades persistidas
5. Apaga uma entidade (Delete)
6. Verifica se uma entidade com determinado ID existe

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);          1

  Optional<T> findById(ID primaryKey);     2

  Iterable<T> findAll();                   3

  long count();                            4

  void delete(T entity);                   5

  boolean existsById(ID primaryKey);       6

  // … more functionality omitted.
}
```

Spring Data PagingAndSortingRepository

- Fornece métodos adicionais para paginação de entidades
- Estende classe CrudRepository
- Exemplo – retorna uma página com 20 objetos

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

Spring Data
Realizando Consultas

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```

- A anotação @Query permite criar consultas específicas.
- Há duas formas de realizar as consultas
  - JPA Query Language (JPQL)
    - Padrão utilizado
  - Native
    - Consulta nativa do banco de dados

Consultas Exemplos

- JPQL

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

- Native

```
@Query( value = "SELECT * FROM USERS u WHERE u.status = 1",
nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

Consultas
Ordenação

- Utilizando métodos disponíveis na classe do tipo Repository

```
userRepository.findAll(Sort.by(Sort.Direction.ASC,
"name"));
```

- JPQL

```
@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort);
userRepository.findAllUsers(Sort.by("name"));
```

Spring Data
Classes Necessárias

- Aplicação
  - Indica que trata-se de uma aplicação baseada no Spring Boot

```
@SpringBootApplication
```

- Controller
  - Classe responsável por tratar as requisições HTTP

```
@Controller
@PostMapping
@GetMapping
```

- Entidade
  - Objeto que é persistido no banco de dados

```
@Entity
```

- Repositório
  - Interface que declara os métodos CRUD disponíveis para a entidade

```
public interface EntRepository extends CrudRepository<Ent, Long>
public interface EntRepository extends JpaRepository<Ent, Long>
```

Docker Linux

- Instalação Docker

sudo apt-get update

sudo apt-get install ca-certificates curl gnupg lsb-release

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

sudo apt install docker.io

echo \ "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \ $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Spring Validation
Validação de Dados

- O Spring fornece suporte para a API Java Bean Validation (https://beanvalidation.org/)
  - https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/
- Esta API fornece uma série de anotações para validação de dados
  - https://beanvalidation.org/2.0/spec/#builtinconstraints
  - Uma outra implementação desta API é feita pelo Hibernate Validator
    - https://hibernate.org/validator/
    - https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/constraints/package-summary.html
- Estas anotações fornecem restrições (constraints) para valores que são recebidos como parâmetros em requisições
- Exemplos de restrições
  - `@NotNull – O valor não pode ser nulo`
  - `@Size(min=n1, max=n2) – O tamanho da string é entre min e max`
- É possível compor as restrições
- No geral as anotações são utilizadas nos atributos das de um Java Bean

Java Bean
Definição

- Um Java Bean é uma classe Java com algumas características próprias
- O java Bean é conhecido como Plain Old Java Object (POJO)
- Características
  - Classe é Serializável
    - Como o Java Bean estende a classe Object no geral já implementa esta interface
  - Classe possui um construtor sem argumentos que é público
  - Classe possui métodos getter e setters para seus atributos
  - Atributos da classe podem ser todos privados
- Utilizando este padrão é possível inspecionar e utilizar a classe utilizado a API de reflexão (reflection) da linguagem Java
  - Esta API permite inspecionar a estrutura de uma classe Java utilizando a própria linguagem (https://docs.oracle.com/javase/tutorial/reflect/)
- No geral um Java Bean contém conceitos ligados ao domínio do problema do software
  - Customer, Product, Contract, ShoppingCart; etc.
- Geralmente o Java Bean é deve ser persistindo pela aplicação

Spring Validation
Anotações - Exemplos

- @NotNull
  - Não permite que o valor seja nulo
- @Min(n2)
  - Valor mínimo para o valor numérico
- @Max(n1)
  - Valor máximo para o valor numérico
- @Size(min=n1, max=n2)
  - Permite string com tamanho mínimo e máximo definidos
- É possível adicionar mensagens de erro específicas utilizando a propriedade message
  - `@NotNull(message="custom message")`
  - `@Size(min=2, max=30, message="tamanho do nome invalido")`
  - `@Size(min=2, max=30, message="{name.prop.invalid}")`

Spring Validation Anotações - Exemplos

- @DateTimeFormat(iso=ISO.***DATE***)
  - ***https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/annotation/DateTimeFormat.html***
- @MaskFormat("(###) ###-####")
  - https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/text/MaskFormatter.html
- @NumberFormat(pattern="$###,###.00")
  - https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/annotation/NumberFormat.html
- @NumberFormat(style=Style.***PERCENT***)
  - ***https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/format/annotation/NumberFormat.Style.html***
- @Future
  - Uma data no futuro
  - https://docs.oracle.com/javaee/6/api/javax/validation/constraints/Future.html
- @Pattern(regexp="^[a-zA-Z0-9]{3}",message="length must be 3")
  - Uma expressão regular
- @Pattern(regexp = "^[a-zA-Z0-9]{6}", message = "Zip Code must be of 6 char/digit")

Spring Validation
Exemplo de Anotações

```
package com.example.validatingforminput;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
public class Person {
  @NotNull @Size(min=2, max=30)
  private String name;
  @NotNull @Min(18) private Integer age;
  public Person (){};
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public Integer getAge() { return age; }
  public void setAge(Integer age) { this.age = age; }
  public String toString() {
    return "Person(Name: " + this.name + ", Age: " + this.age + ")";
  }
}
```

Spring Validation
Entidade Exemplo

```
package com.example.accessingdatajpa;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity

public class Customer {

  @Id

  @GeneratedValue(strategy=GenerationType.AUTO)

  private Long id;  private String firstName;  private String lastName;

  protected Customer() {}

  public Customer(String firstName, String lastName) {this.firstName = firstName;   this.lastName = lastName;   }

  @Override

  public String toString() {return String.format("Customer[id=%d, firstName='%s', lastName='%s']",id, firstName,
lastName);

  }
 //getters e setters

}
```

Spring Validation
Integração com Controller

- A anotação **@Valid** verifica se os parâmetros recebidos são válidos
- Esta anotação permite que o Spring realize a validação dos parâmetros
- Caso a validação falhe o método dispara uma exceção do tipo `MethodArgumentNotValidException`
- É possível adicionar no controlador um método para trata as exceções. Este método deve conter a anotação **@ExceptionHandler**

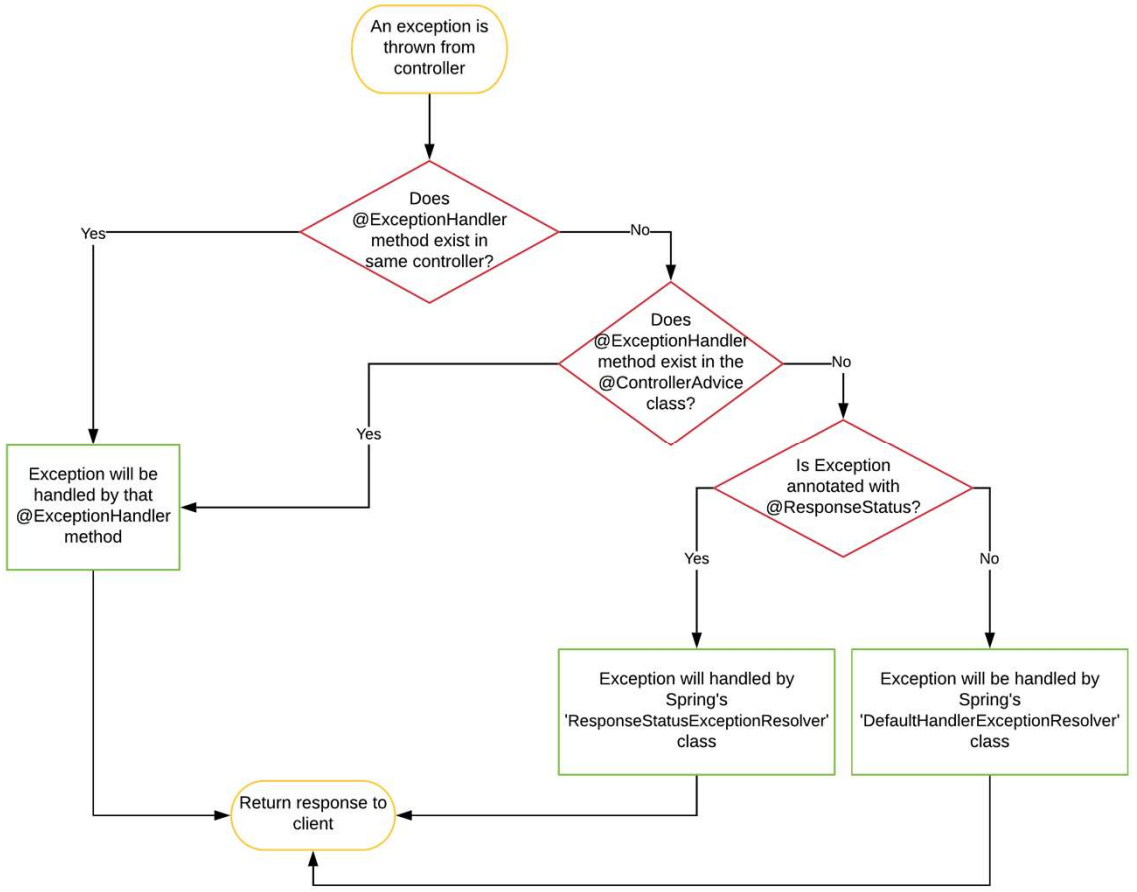
Spring Validation
Integração com Controller

```
@RestController
public class UserController {
  @PostMapping("/users")
  ResponseEntity<String> addUser(@Valid @RequestBody User user) {
     // logic to persist the user
     return ResponseEntity.ok("User is valid");
  }

}
```

Spring Boot Validation
Tratamento de Exceções

- O mecanismo de tratamento de exceções do Spring Boot suporta diferentes formas para tratar as exceções

- @ExceptionHandler
- @ExceptionHandler em uma classe com anotação @ControllerAdvice
  - Pode envolver a anotação @ResponseSatus ou não



Tratamento de Exceções Diretamente no Controller

- É feito com a anotação @ExceptionHandler

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleMethodArgumentNotValid(MethodArgumentNotValidException ex) {
Map<String, String> errors = new HashMap<>();
ex.getBindingResult().getFieldErrors().forEach(error ->
errors.put(error.getField(), error.getDefaultMessage()));
return errors;
}
```

- O Código acima permite tratar exceções do tipo **MethodArgumentNotValidException**
- A anotação **ExceptionHandler** indica o método que trata um determinado tipo de exceção
- Um objeto do tipo **BindingResult** encapsula as mensagens de erro retornadas na exceção
- O método pode ser utilizado para retornar no objeto HTTP as mensagens de erro em um formato JSON

Spring Validation
Customizando as Execeções

- A anotação @ControllerAdvice indica que classe que trata as exceções estará disponível para todos os controllers da aplicação
- Assim é possível centralizar nesta classe todos métodos de tratamento de exceção em uma aplicação Spring
- Neste caso todo o código responsável pelo tratamento destas exceções fica em um único ponto

CustomExceptionHandler

```
@SuppressWarnings({"unchecked","rawtypes"})
@ControllerAdvice
public class CustomExceptionHandler extends ResponseEntityExceptionHandler {
//neste caso é possível adicionar nesta classe todos os outros métodos
@Override protected ResponseEntity<Object>
handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
HttpHeaders headers, HttpStatus status, WebRequest request) {
    List<String> details = new ArrayList<>();
    for(ObjectError error : ex.getBindingResult().getAllErrors()) {
      details.add(error.getDefaultMessage());
    }
    ErrorResponse error = new ErrorResponse("Validation Failed", details);
    return new ResponseEntity(error, HttpStatus.BAD_REQUEST);
  }
  //outros métodos para tramento de exceção
}
```

Spring Data + Validation
Classes Necessárias

- Aplicação
  - Indica que trata-se de uma aplicação baseada no Spring Boot

```
@SpringBootApplication
```

- Controller
  - Classe responsável por tratar as requisições HTTP

```
@Controller
@PostMapping
@GetMapping
@Valid
@ExceptionHandler
```

- Entidade
  - Objeto que é persistido no banco de dados

```
@Entity
@Contraints
```

- Repositório
  - Interface que declara os métodos CRUD disponíveis para a entidade

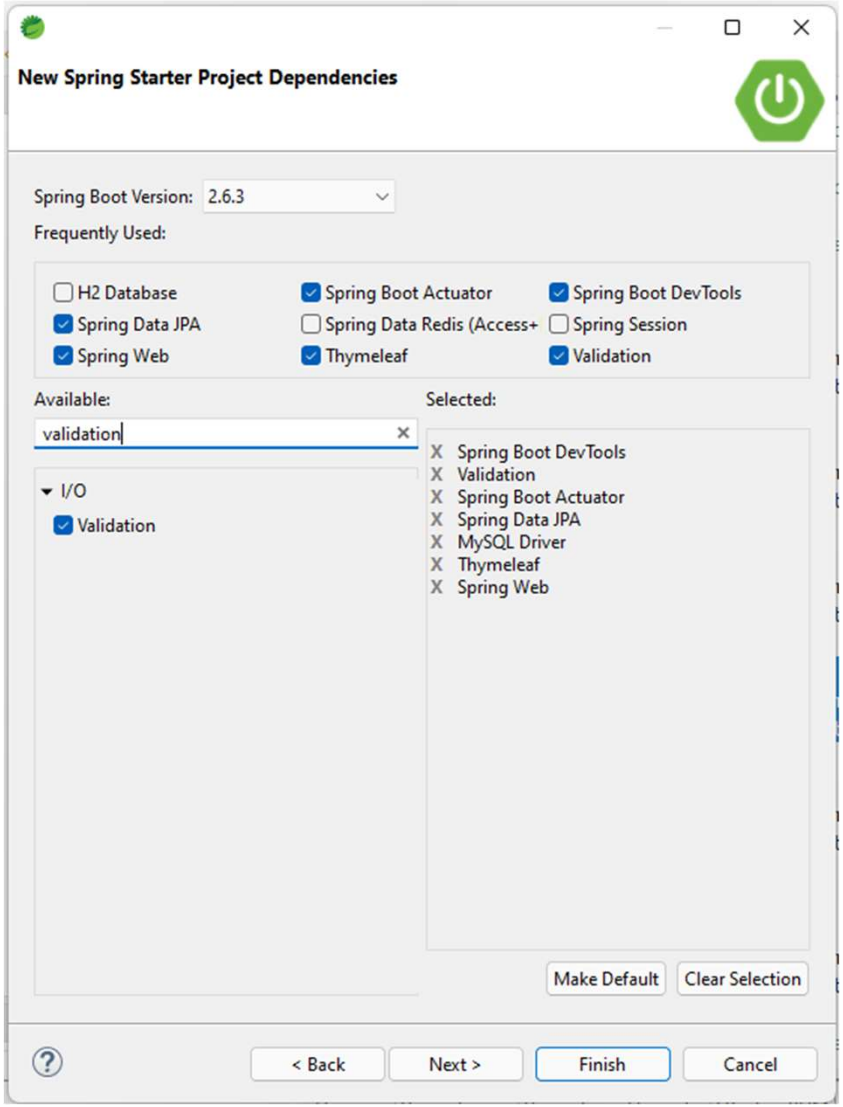
```
public interface EntRepository extends CrudRepository<Ent, Long>
public interface EntRepository extends JpaRepository<Ent, Long>
```

Spring Componente para Validação

- A validação no Spring envolve o seguinte componente

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- O componente de I/O pode ser selecionado no Spring Starter Project ou no Spring Initializr
- No exemplo ao lado trata-se de um projeto que envolve
  - Acesso a dados (MySQL utilizando JPA)
  - Aplicação Web
  - Validação de Dados

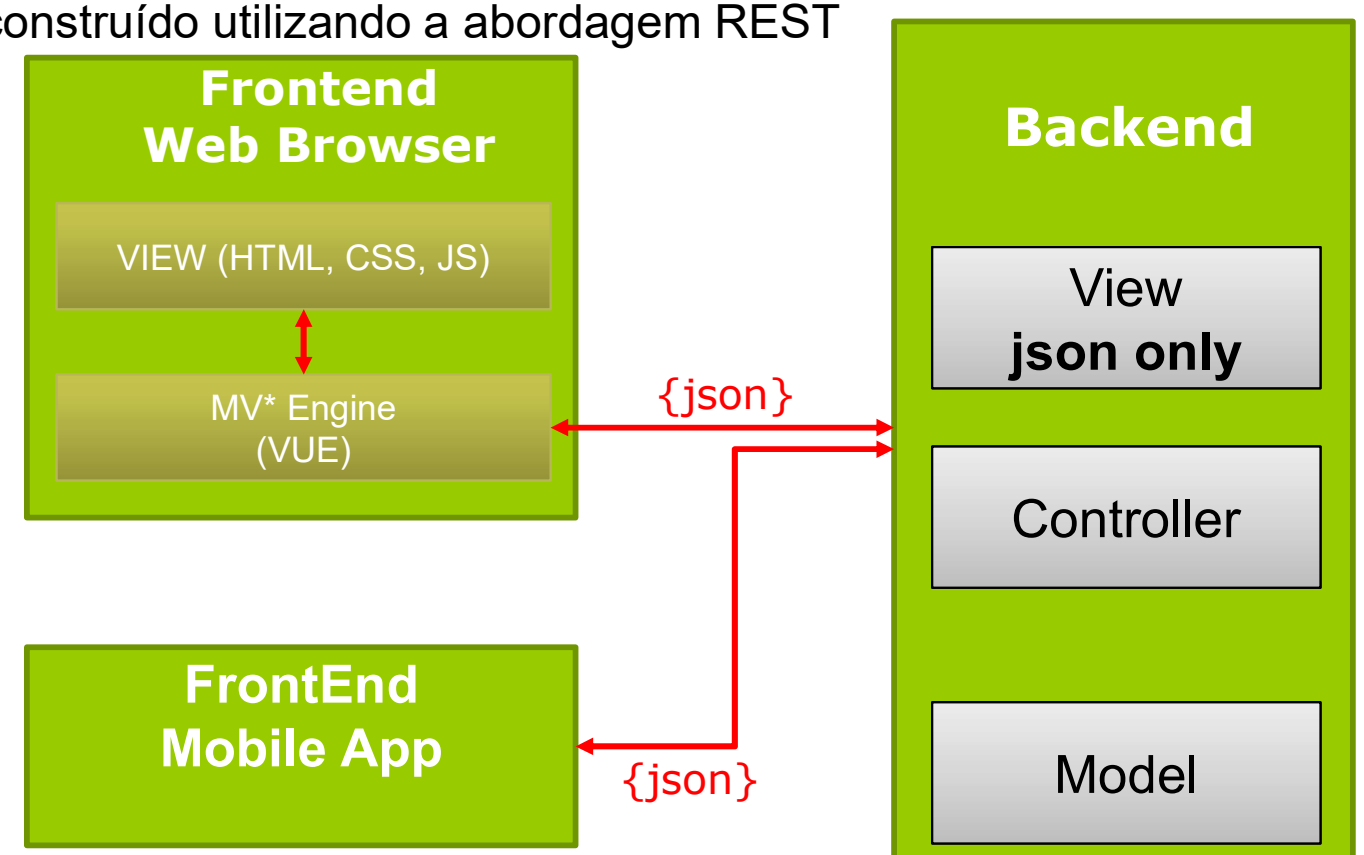


Representational state transfer (REST)

- REST é um estilo arquitetural para a criação de serviços baseado na web
- Como é baseado na web, o protocolo para comunicação é o HTTP
- Algumas características dos serviços REST
  - Interface Simples e Consistente
  - Não armazenam estado (stateless)
  - Representação padronizada dos objetos (resources) – JSON/XML
  - Facilmente Escaláveis
- https://en.wikipedia.org/wiki/Representational_state_transfer
- Abordagem REST é a mais utilizada nas aplicações Web mais recentes
  - Serviços são expostos e podem ser consumidos tanto por clientes web (navegadores) como por aplicações móveis
- No geral serviços baseado em REST não transportam conteúdo HTML, mas objetos baseados no formato **JavaScript Object Notation** (JSON)
  - Formato padronizado para troca de dados entre aplicações
  - Baseado em Texto com pares de atributo-valor com a serialização de objetos complexos
  - O content-type para o conteúdo padronizado é `application/json`

Model-View-* (frontend) + REST(backend)

- Uma estratégia é utilizar no Frontend web alguma engine MV* para preparar o conteúdo para ser exibido no browser e realizar ligação entre o backend e o que é visto no Frontend pelo usuário
- Neste caso uma representação neutra e livre de marcações HTML/CSS, o JSON, é devolvida e processada corretamente por cada plataforma de Frontend (mobile ou web)
- O backend por sua vez é construído utilizando a abordagem REST



JavaScript Object Notation (JSON) Exemplo

- A representação serializa (coloca em série) todas as propriedades do objeto
- class Person
  - String firstName;
  - String lastName;
  - boolean isAlive;
  - int age;
  - Address address;
  - ArrayList<Person> children;
  - Person spouse;
- Class Adress
  - String streetAddress;
  - String city;
  - String state;
  - String postalCode;
  - ArrayList<Phone> phoneNumbers;
- Class Phone
  - String type;
  - String number;

```
{
"firstName": "John",
"lastName": "Smith",
"isAlive": true,
"age": 27,
"address": {
   "streetAddress": "21 2nd Street",
   "city": "New York",
   "state": "NY",
   "postalCode": "10021-3100" },
   "phoneNumbers": [
      {
         "type": "home",
         "number": "212 555-1234"},
      {
          "type": "office",
          "number": "646 555-4567"
      }
   ],
   "children": [],
   "spouse": null
}
```

Spring
Serviços Baseados em REST

- O serviços baseados no conceito de REST utilizam os métodos do protocolo HTPP que são mapeados em serviços.
- O serviços contém uma lógica e são executados no lado servidor
- Um mapeamento básico que pode ser feito é com os métodos do protocolo HTTP com as operações CRUD básicas de persistência

HTTP	CRUD	EXEMPLO DO MÉTODO HTTP
POST	**C**REATE	http://serverName:port/api/entity/{id}
GET	**R**EAD	http://serverName:port/api/entity/{id} http://serverName:port/api/entity
PUT	**U**PDATE	http://serverName:port/api/entity/{id}
DELETE	**D**ELETE	http://serverName:port/api/entity/{id}

Spring REST

- @RestController
  - Esta anotação indica que o controller deverá retornar os dados diretamente no corpo da resposta sem a necessidade utilizar um template
- Para cada operação HTTP há uma anotação específica
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
- No Controller é necessário mapear para método do protocolo HTTP qual será o serviço (método da classe controller) que vai responder
- O Spring também aceita a seguinte anotação
  - @RequestMapping(method = RequestMethod.GET)
  - @RequestMapping(method = RequestMethod.POST)
  - @RequestMapping(method = RequestMethod.PUT)
  - @RequestMapping(method = RequestMethod.DELETE)

REST Controller
Exemplo Básico

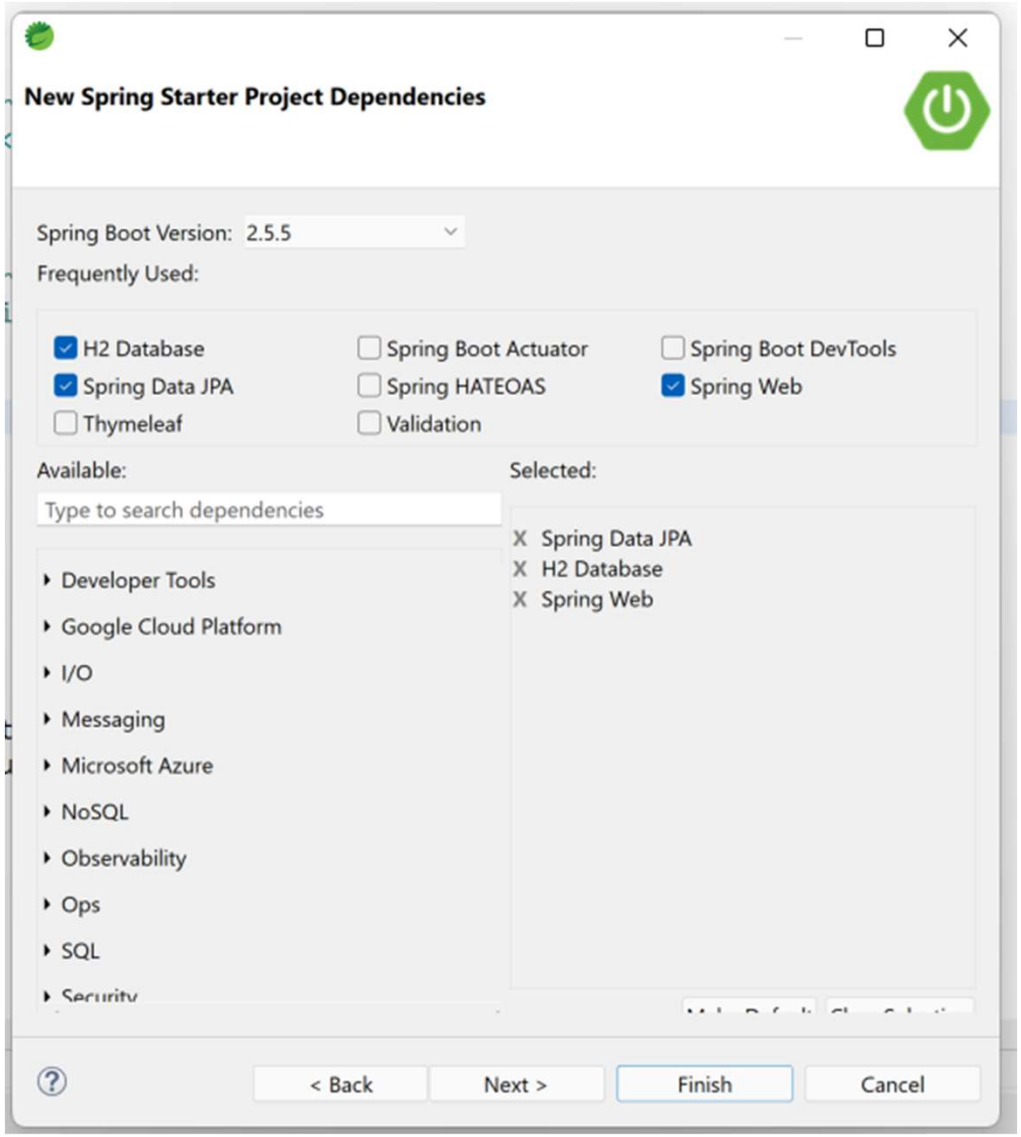
- Este exemplo apresenta um controller básico para criar um serviço RESTFull no Spring
- No exemplo é utilizada a persistência em memória com a base H2
  - H2 é escrito em Java e Spring Boot oferece suporte para embutir este banco de dados na aplicação
  - Ao criar o projeto no Spring Initializr é necessário incluir o suporte ao H2
- A aplicação contém as seguintes classes:
  - Domain Model
    - Employee
      - Entidade (resource) manipulada pela aplicação e que será persistida
  - Persistência
    - EmployeeRepository
      - Interface que estende JpaRepository e define as operações de persistência
    - EmployeeLoadDatabase
      - Classe auxiliar para carregar dados em memória durante a inicialização da aplicação
  - Controller
    - EmployeeSimpleController
      - Responsável por mapear as requisições HTTP nos serviços java que estão implantados no lado servidor

Anotações

- @Autowired
  - Permite a injeção de dependência (DI) em uma aplicação baseada no Spring Boot
  - Permite incluir uma classe (bean) em uma outra classe durante a execução
- @Configuration
  - Classe utilizada pela aplicação Spring Boot que é responsável por carregar para o contexto da aplicação as entidades (@Bean)
  - Esta classe permite alterar a configuração da aplicação no momento de carregamento
  - Um exemplo seria uma classe para configurar um acesso a banco de dados
- @Bean
  - Objeto gerenciado pelo Spring e que é injetado no contexto da aplicação para seu funcionamento e configuração
- Classes/Interface de Suporte
  - CommandLineRunner
    - Esta interface permite executar alguma lógica antes do inicio da aplicação pelo Spring
    - No exemplo uma classe (EmployeeLoadDatabase) cria um bean do tipo CommandLineRunner
    - O código associado é executado no momento de inicialização da aplicação

Spring
Serviços Baseados em REST

- Componentes Básicos
  - Spring Web
    - Suporte ao @RestController
  - Spring Data JPA
    - Suporte à persistência de dados
  - H2
    - Banco de dados H2



REST Controller
Exemplo Básico

```
package com.example.modernREST;
@RestController
@RequestMapping("/api/employee")
public class EmployeeSimpleController {
    @Autowired
    private EmployeeRepository employeeRepository;
    @GetMapping
    public List<Employee> findAllEmployees() {
     // Service Logic
    }
    @GetMapping("/{id}")
    public ResponseEntity<Employee> findEmployeeById(@PathVariable(value = "id") long id) {
     // Service Logic
    }
    @PostMapping
    public Employee saveEmployee(@Validated @RequestBody Employee employee) {
     // Service Logic
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<String> deleteEmployeeById(@PathVariable(value = "id") long id) {
     // Service Logic
    }
}
```

REST API
Entidade (Employee)

```
package com.example.modernREST;
import ...
@Entity
class Employee {
  private @Id @GeneratedValue Long id; private String name; private String role;
  Employee() {}
  Employee(String name, String role) {this.name = name; this.role = role;}

 //getters and setters
  @Override
  public boolean equals(Object o) {
    //implmentation
  }
  @Override
  public int hashCode() {
     //implementation
  }
  @Override
  public String toString() {
     //implementation
  }
}
```

REST API
Repositório JPA

```
package com.example.modernREST;

import org.springframework.data.jpa.repository.JpaRepository;

interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

REST API
Classe Apoio – Carregar BD Memória

```
package com.example.modernREST;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class EmployeeLoadDatabase {

  private static final Logger log = LoggerFactory.getLogger(EmployeeLoadDatabase.class);

  @Bean
  CommandLineRunner initDatabase(EmployeeRepository repository) {

    return args -> {
      log.info("Preloading " + repository.save(new Employee("Bilbo Baggins", "burglar")));
      log.info("Preloading " + repository.save(new Employee("Frodo Baggins", "thief")));
    };
  }
}
```

REST API
Controller – Código Completo

```
package com.example.modernREST;
import ...
@RestController
@RequestMapping("/api/employee")
public class EmployeeSimpleController {
    @Autowired
    private EmployeeRepository employeeRepository;
    @GetMapping
    public List<Employee> findAllEmployees() {
      return employeeRepository.findAll();
    }
    @GetMapping("/{id}")
    public ResponseEntity<Employee> findEmployeeById(@PathVariable(value = "id") long id) {
        Optional<Employee> employee = employeeRepository.findById(id);
        if(employee.isPresent()) {
            return ResponseEntity.ok().body(employee.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }
```

REST API
Controller – Código Completo

```
    @PostMapping
    public Employee saveEmployee(@Validated @RequestBody Employee employee) {
       return employeeRepository.save(employee);
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<String> deleteEmployeeById(@PathVariable(value = "id") long id) {
        Optional<Employee> employee = employeeRepository.findById(id);
        if(employee.isPresent()) {
        employeeRepository.deleteById(id);
            return ResponseEntity.ok().body("Deleted");
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Hypermedia as the Engine of Application State (HATEOAS)

- Uma aplicação REST que segue o “padrão” HATEOAS devolve informações adicionais através hyperlinks que permite ao cliente ter maior Informação sobre quais operações pode executar sobre um determinado objeto (resource)
- Com o HATEOAS o cliente fica mais desacoplado do servidor e o servidor pode evoluir a funcionalidade e o cliente pode descobrir como consumir estas novas funcionalidades
- O servidor devolve informação de forma dinâmica para o cliente utilizando o conceito de Hypermedia (neste caso os hyperlinks)
- Com estes links o cliente recebe informações sobre como pode interagir posteriormente com aquele resource
- Os links da resposta podem variar conforme o estado do resource
- No JSON da resposta os diversos links são devolvidos em um formato padrão

```
"links": {
        “linkname": “hyperlinkAssociatedwithResource",
}
```

- O formato padrão para representar resources e links é conhecido como Hypertext Application Language (HAL)
- No HATEOAS a resposta contém JSON+HAL (extra links)

HATEOAS Rest Service Exemplo

- Um exemplo de consulta REST

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
```

- Resposta de um serviço construído com o princípio HATEOAS

```
HTTP/1.1 200 OK

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposits": "/accounts/12345/deposits",
            "withdrawals": "/accounts/12345/withdrawals",
            "transfers": "/accounts/12345/transfers",
            "close-requests": "/accounts/12345/close-requests"
        }
    }
}
```

Dados

Hypermedia

HATEOAS Rest Service Exemplo

- A mesma consulta em um momento posterior

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
```

- Neste caso a resposta indica que a única operação possível é realizar o depósito, indicando que a conta não possui mais saldo
- A resposta muda conforme o estado da aplicação

```
HTTP/1.1 200 OK

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": -25.00
        },
        "links": {
            "deposits": "/accounts/12345/deposits"
        }
    }
}
```

Spring HATEOAS

- O Spring oferece suporte ao princípio HATEOAS diretamente no Spring Initializr

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

- Classes (e Interfaces) de Suporte
  - Link
    - O link pode ser um URI ou um template para uma URI
  - RepresentationModel
    - Classe Genérica e que possui como subclasses EntityModel e CollectionModel
    - Container para coleção de objetos da classe Link
    - Métodos para manipular link
  - EntityModel
    - Classe que representa uma entidade particular e os links associados
  - CollectionModel
    - Classe que presenta uma coleção de entidades
    - Pode ser utilizada nos métodos que devolvem uma coleção de Objetvos
  - RepresentationModelAssembler
    - Interface com assinatura dos métodos para converter um entidade do domínio em um RepresentationModel daquela entidade
    - Necessário incluir uma classe que implementa esta interface e fornece a lógica de conversão de um objeto do domínio (Entity) e uma representação dentro do padrão HATEOAS (EntityModel)
  - WebMvcLinkBuilder
    - Esta classe fornece o suporte para criar o links sem a necessidade de incluir a lógica de criação no código

Spring HATEOAS

- Considerando uma classe, por exemplo Entidade Employee, que está lida com a persistência
- Utilizando o suporte do Spring HATEOAS a Entidade (Entity) será convertida para o seu RepresentationModel
  - Para isto RepresentationModelAssembler e WebMvcLinkBuilder são utilizados
  - Neste exemplo será criada a classe EmployeeModelAssembler
  - Isto é feito com sobrecargas de métodos
- Este processo é realizado para todos os serviços ou entidades envolvidos

Exemplo EmployeeModelAssembler

```
package com.example.modernREST;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.server.RepresentationModelAssembler;
import org.springframework.stereotype.Component;

@Component
class EmployeeModelAssembler implements RepresentationModelAssembler<Employee,
EntityModel<Employee>> {

@Override
public EntityModel<Employee> toModel(Employee employee) {

return EntityModel.of(employee, //
linkTo(methodOn(EmployeeController.class).one(employee.getId())).withSelfRel(),
linkTo(methodOn(EmployeeController.class).all()).withRel("employees"));
}
}
```

REST API
Controller HATEOAS – Código Completo

```
package com.example.modernREST;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
import java.util.List;
import java.util.stream.Collectors;
import org.springframework.hateoas.CollectionModel;
import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.IanaLinkRelations;
import org.springframework.http.ResponseEntity;

@RestController
@RequestMapping("/api2")
public class EmployeeController {

    private final EmployeeRepository repository;
```

REST API
Controller HATEOAS – Código Completo

```
@RestController
@RequestMapping("/api2")
public class EmployeeController {
  private final EmployeeRepository repository;
  private final EmployeeModelAssembler assembler;

  EmployeeController(EmployeeRepository repository, EmployeeModelAssembler assembler) {
    this.repository = repository;
    this.assembler = assembler;
  }

  @GetMapping("/employees")
   CollectionModel<EntityModel<Employee>> all() {
     List<EntityModel<Employee>>
     employees = repository.findAll().stream().map(assembler::toModel).collect(Collectors.toList());

     return CollectionModel.of(employees, linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
    }
```

```
@PostMapping("/employees")
ResponseEntity<?> newEmployee(@RequestBody Employee newEmployee) {
```

REST API
Controller HATEOAS – Código Completo

```
  @PostMapping("/employees")
   ResponseEntity<?> newEmployee(@RequestBody Employee newEmployee) {
     EntityModel<Employee> entityModel = assembler.toModel(repository.save(newEmployee));
  return
ResponseEntity.created(entityModel.getRequiredLink(IanaLinkRelations.SELF).toUri()).body(entityModel);
}

  @GetMapping("/employees/{id}")
   EntityModel<Employee> one(@PathVariable Long id) {
     Employee employee = repository.findById(id).orElseThrow(() -> new EmployeeNotFoundException(id));
     return assembler.toModel(employee);
   }

@PutMapping("/employees/{id}")
ResponseEntity<?> replaceEmployee(@RequestBody Employee newEmployee, @PathVariable Long id) {
  Employee updatedEmployee = repository.findById(id).map(employee -> {
                                                    employee.setName(newEmployee.getName());
                                                    employee.setRole(newEmployee.getRole());
                                                    return repository.save(employee);
                                          }).orElseGet(() -> {
                                                                    newEmployee.setId(id);
                                                                    return repository.save(newEmployee);
```

REST API
Controller HATEOAS – Código Completo

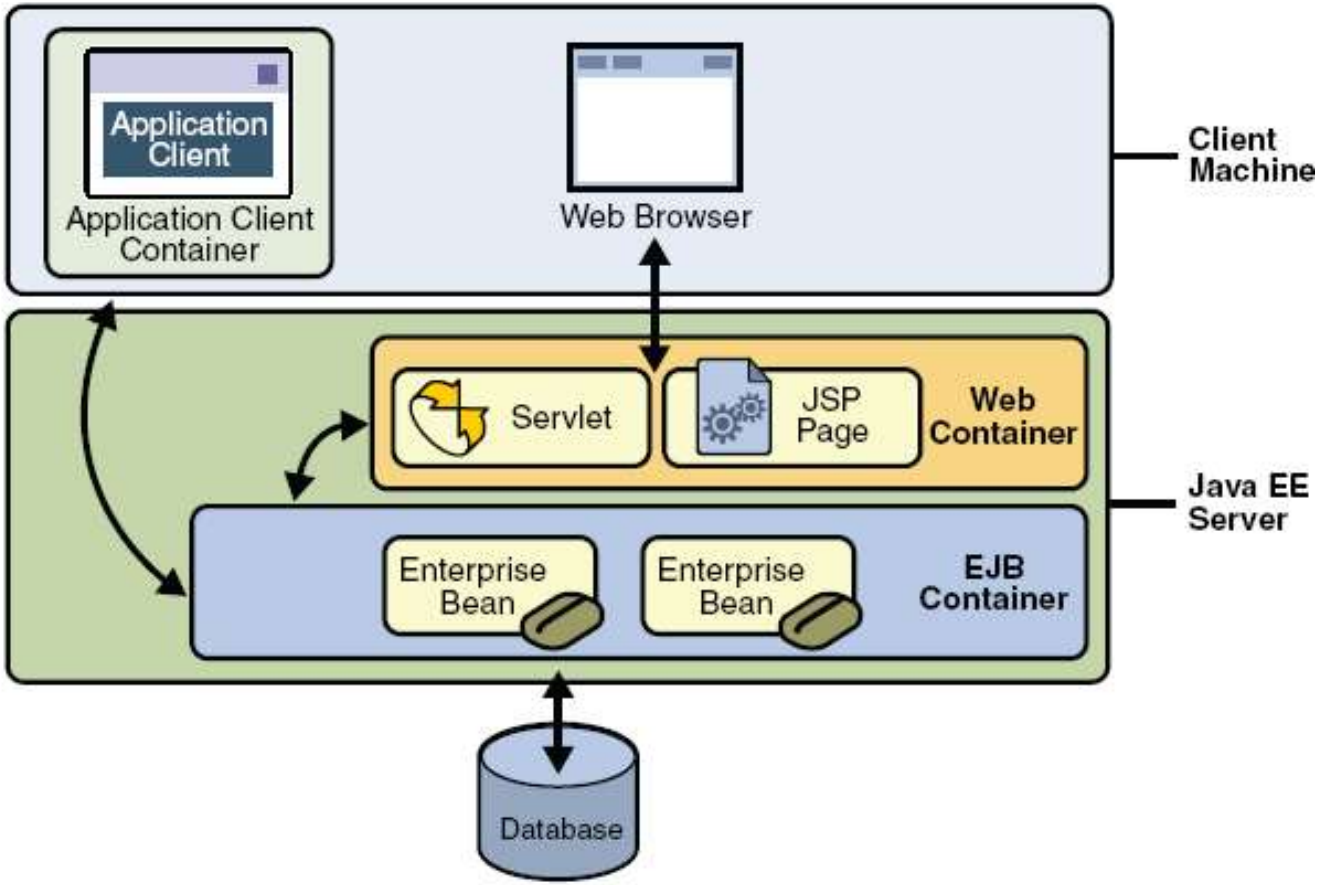
```
   @PutMapping("/employees/{id}")
ResponseEntity<?> replaceEmployee(@RequestBody Employee newEmployee, @PathVariable Long id) {
    Employee updatedEmployee = repository.findById(id).map(employee -> {
          employee.setName(newEmployee.getName());
          employee.setRole(newEmployee.getRole());
          return repository.save(employee);
        }).orElseGet(() -> {
          newEmployee.setId(id);
          return repository.save(newEmployee);
        });
        EntityModel<Employee> entityModel = assembler.toModel(updatedEmployee);
        return
ResponseEntity.created(entityModel.getRequiredLink(IanaLinkRelations.SELF).toUri()).body(entityModel);
                }

@DeleteMapping("/employees/{id}")
  ResponseEntity<?> deleteEmployee(@PathVariable Long id) {
     repository.deleteById(id);
     return ResponseEntity.noContent().build();
   }
}
```

Sessões em Aplicações Web

- O protocolo HTTP é **stateless**, ou seja, não existe para o protocolo a manutenção do estado da troca de mensagens
- Um conceito comum em aplicações web é o conceito de Sessão de usuário
- Na sessão do usuário o estado da aplicação é mantido ao longo do tempo e informações são compartilhadas entre diferentes requisições
- Como o protocolo HTTP não possui este suporte os servidores de aplicações baseadas em HTTP possuem esta funcionalidade
- Através de Sessão é possível por exemplo:
  - Manter um carrinho de compras em uma aplicação de e-commerce
  - Realizar o login de usuário em uma aplicação bancária: a sua agência bancária e conta permanecem inalteradas durante a sessão
  - E muito outros exemplos...
- A plataforma Java possui fornece um ambiente de execução para aplicações web dinâmica
  - Este ambiente é chamado Web contêiner

Java EE contêineres

- O contêiner fornece a infra-estrutura para a execução de um componente específico da plataforma
- O servidor Java J2EE (Application Server) fornece dois tipos básicos de contêineres:
  - WEB contêiner
  - EJB contêiner



Java EE contêineres
Web Contêiner

- Executa Servlets e páginas JSP
- Servlets
  - São classes Java que podem ser chamados dinamicamente e que podem executar alguma funcionalidade.
  - Estes programas podem ser executados em um Servidor Web ou em um contêiner para Servlets
  - Normalmente estão ligados a geração de conteúdo para browsers.
  - O Servlet implementa a interface Servlet e possui um funcionamento pré-definido
  - Os Servlets recebem e respondem a requisições feitas normalmente através do protocolo HTTP
- JavaServer Pages (JSP)
  - Consistem de uma maneira para criar conteúdo dinâmico para a Web
  - Seu objetivo é criar uma separação entre a apresentação e os dados que estarão presentes em uma página no navegador.
  - Normalmente a página JSP é um modelo que contém tanto o conteúdo estático, como a indicação de como o conteúdo dinâmico será gerado

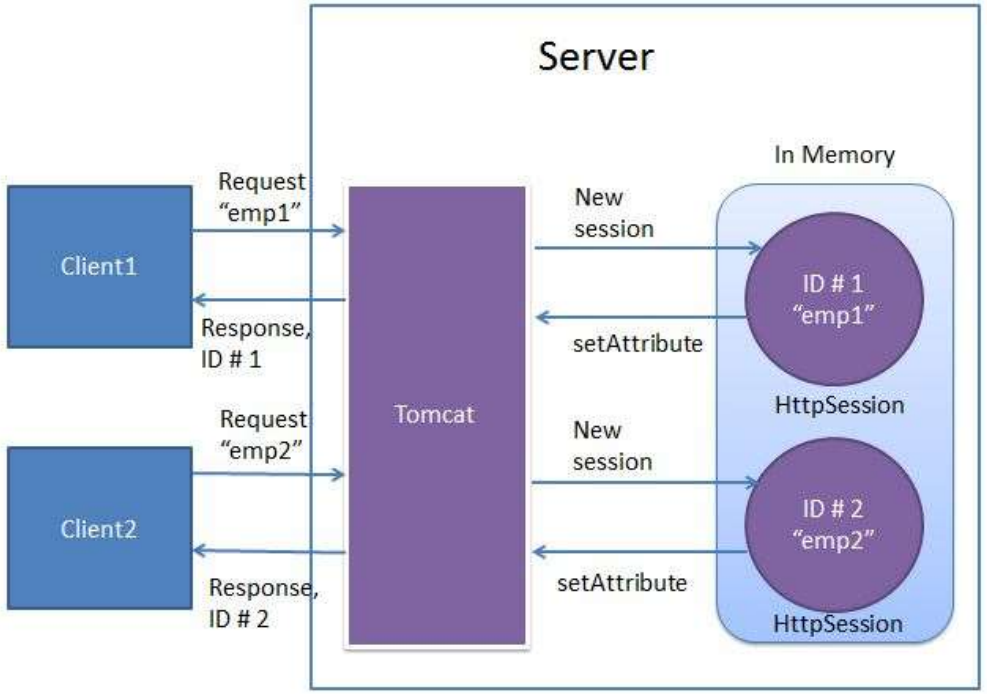
Apache Tomcat

- O apache Tomcat basicamente é um contêiner Web (Servlets e JSP)
- O Tomcat é a implementação de referência das especificações para Servlets e JavaServer Pages
- Estas especificações são desenvolvidas pela Sun através da JCP (Java Community Process).
- A JCP é uma entidade aberta que possui como membros pessoas e empresas envolvidas com a tecnologia Java.
- A seguintes tabela mostra as versões das especificações e as respectivas versões do Tomcat

Especificações Servlet/JSP	Versão Apache Tomcat	Versão Atual	Versão Miníma Java SE
6.0/3.1	10.1.x	10.1.0-M6	11
5.0/3.0	10.0.x	10.0.12	8
4.0/2.3	9.0.x	9.0.54	1.6
3.0/2.2	7.0.x	7.0.21	1.6
2.5/2.1	6.0.x	6.0.32	1.5
2.4/2.0	5.5.x	5.5.33	1.4
2.3/1.2	4.1.x	4.1.40	1.3
2.2/1.1	3.3.x	3.3.2	1.1

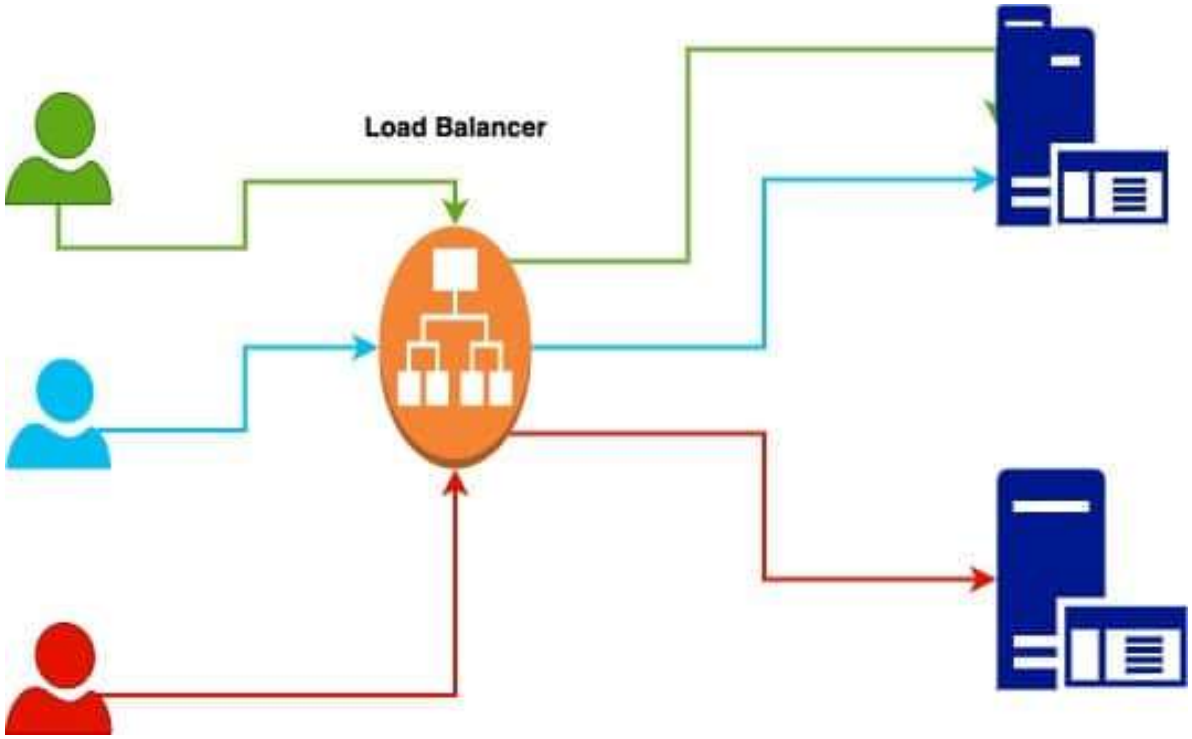
Sessões em Aplicações Web Java Tomcat

- O Tomcat possui suporte para Sessões
- Dados da Sessão são armazenados em memória
- Uma falha do Tomcat e todos os dados da sessão são perdidos
- Esta solução também apresenta um problema de escalabilidade
- Uma estratégia é ter várias instâncias do Tomcat



Sessões em Aplicações Web Java Tomcat - LoadBalancer

- Neste caso é necessário um LoadBalancer que fazer o papel de rotear um mesmo usuário para uma mesma instância do Tomcat



Spring Session

- Fornece suporte para gestão de uma sessão do usuário em aplicações baseadas no Spring
  - https://docs.spring.io/spring-session/reference/
- Tipos de Sessões
  - HttpSession
    - substituir a HttpSession em um contêiner de aplicação (como o Tomcat) de maneira neutra
    - Suporte IDs de sessão em cabeçalhos para trabalhar com APIs RESTful
  - WebSocket
    - fornece a capacidade de manter a HttpSession ativa ao receber mensagens WebSocket
  - WebSession
    - Substituir a WebSession do Spring WebFlux de uma forma neutra do container de aplicação

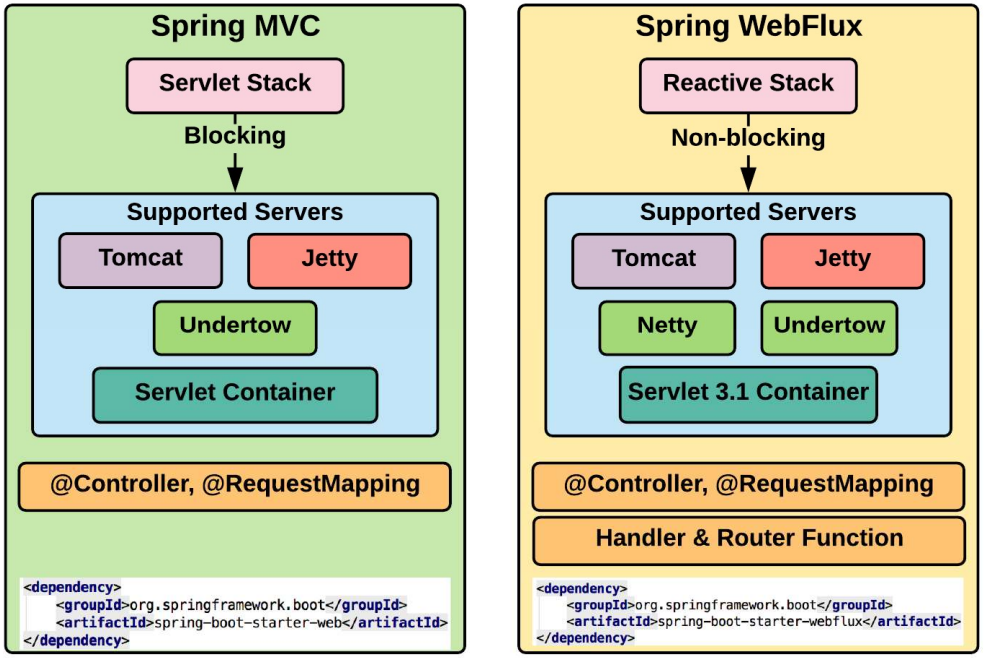
Spring Session

- O projeto Spring Session fornece o suporte para incluir a gestão de sessão pelo Spring
- Dependência para adicionar o Spring Session

```
<dependency>
<groupId>org.springframework.session</groupId>
<artifactId>spring-session-core</artifactId>
</dependency>
```

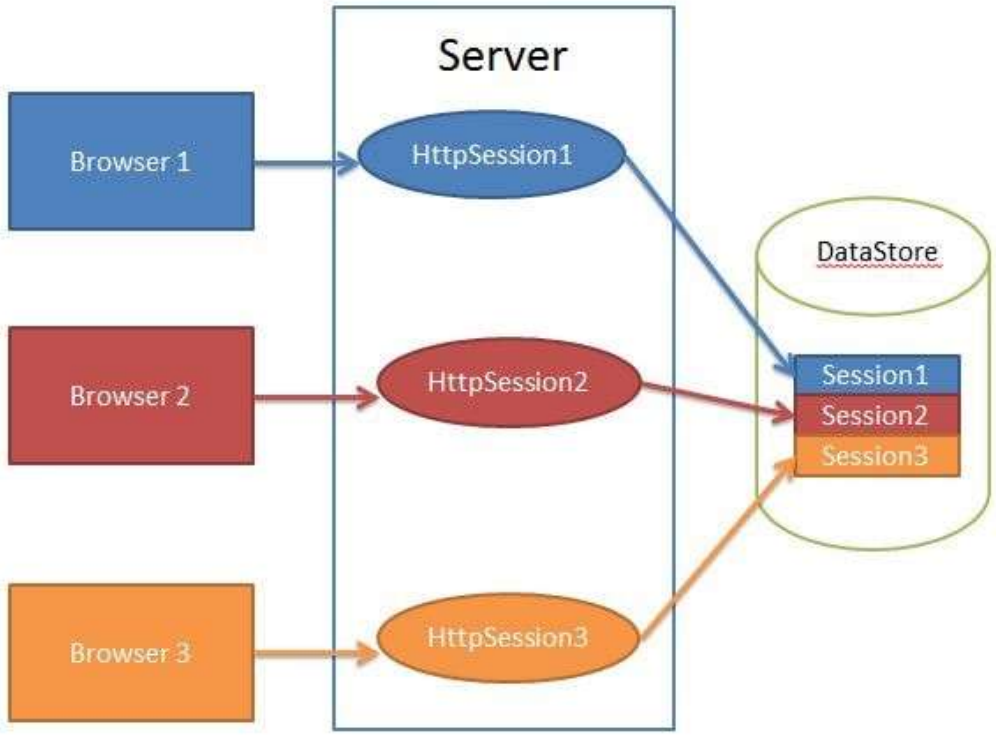
Spring WebFlux

- Suporta programação reativa via Reactive Stream
- Utilizado para construir aplicações web, o processamento neste caso é assíncrono e não bloqueante.
- Comparação Spring MVC x Spring WebFlux



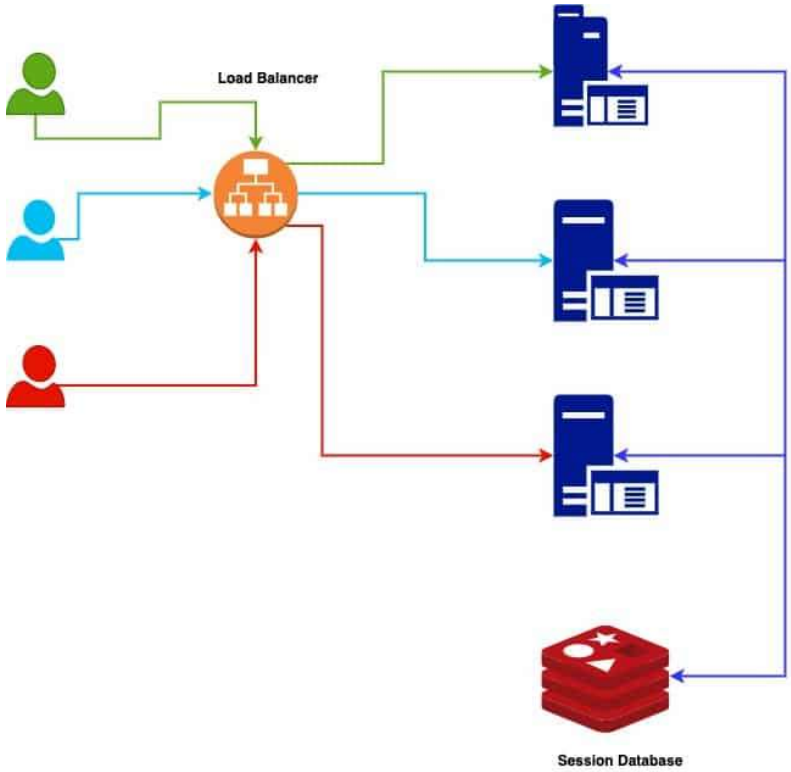
Sessões em Aplicações Web Java Spring Session

- O Spring Session fornece suporte para diferentes estratégias de persistência dos dados da Sessão
  - JDBC
  - Redis (https://redis.io/)
  - Hazelcast
  - MongoDB
  - Apache Geode



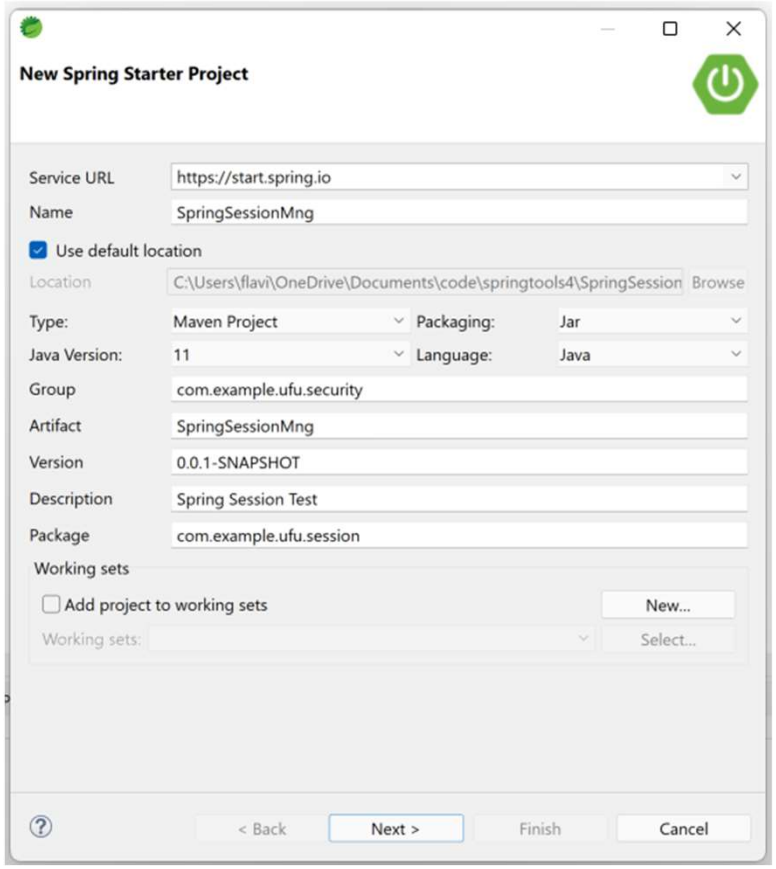
Sessões em Aplicações Web Java Spring Session

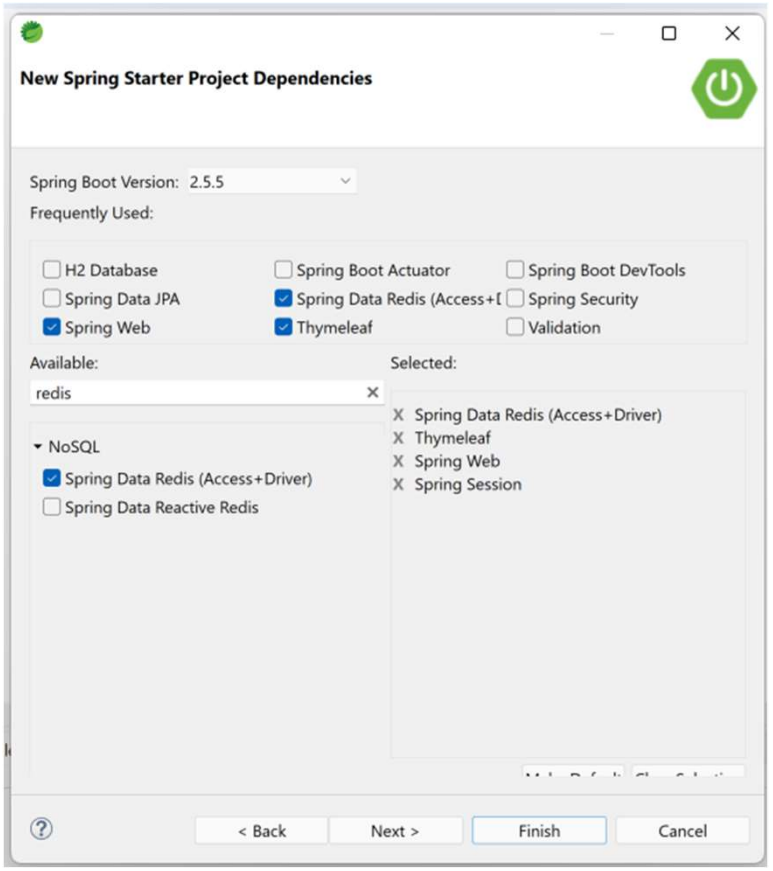
- Neste caso várias instâncias da aplicação Spring possuem sua sessão armazenada no banco de dados
- Neste caso será utilizado o REDIS
  - Banco de dados open source
  - Diversas Estruturas de dados em memória
    - String, hashes, lists, set, sorted sets



Spring Session

- Projetos Spring utilizados
  - Spring web
  - Spring Session
  - Spring Data Redis (Access + Driver)
  - Thymeleaf





Spring Session REDIS

- Para utilizar o Spring Session é necessário indicar o tipo de persistência que será utilizado
- Isto pode ser feito no arquivo “application.properties”
  - https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html
- No exemplo abaixo a configuração utilizar o banco de dados REDIS
  - `spring.session.store-type=redis`
- Outros parâmetros para o REDIS
    - `spring.redis.host=172.24.119.180`
    - `spring.redis.port=6379`
  - Configurações adicionais:
    - `spring.redis.password=` `#password`
    - `server.servlet.session.timeout=` `# Session timeout.`
    - `spring.session.redis.flush-mode=on-save` `# Sessions flush mode.`
    - `spring.session.redis.namespace=spring:session` `# Namespace for keys used to store sessions.`
- Para outros tipos de banco dados
  - JDBC - `spring.session.store-type=jdbc`
  - Hazelcast - `spring.session.store-type=hazelcast`
  - MongoDB - `spring.session.store-type=mongodb`
  - Para cada tipo de banco de dados há configurações específicas como mostrado para o REDIS

Spring Session
Controller Exemplo 1/4

```
package com.example.ufu.session;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
```

Spring Session Controller Exemplo 2/4

```
@Controller
public class SpringSessionController {
   @GetMapping("/")
   public String process(Model model, HttpSession session) {
   @SuppressWarnings("unchecked")
   List<String> messages = (List<String>) session.getAttribute("MY_SESSION_MESSAGES");
   if (messages == null) {
    messages = new ArrayList<>();
   }
   model.addAttribute("sessionMessages", messages);
   model.addAttribute("sessionId", session.getId());
   User storedUser = (User) session.getAttribute("SESSION_USER_DATA");
   if (storedUser != null) {
      model.addAttribute("userData", storedUser);
   }
   return "index";
}
```

Spring Session Controller Exemplo 3/4

```
@PostMapping("/persistMessage")
public String persistMessage(@RequestParam("msg") String msg, HttpSession session) {
   @SuppressWarnings("unchecked")
   List<String> messages = (List<String>) session.getAttribute("MY_SESSION_MESSAGES");
   if (messages == null) {
      messages = new ArrayList<>();
      session.setAttribute("MY_SESSION_MESSAGES", messages);
   }
   messages.add(msg);
   session.setAttribute("MY_SESSION_MESSAGES", messages);
   return "redirect:/";
}
@PostMapping("/login")
public String saveUserData(User user, Model model, HttpSession session) {
   @SuppressWarnings("unchecked")
   User storedUser = (User) session.getAttribute("SESSION_USER_DATA");
   if (storedUser == null) {
      session.setAttribute("SESSION_USER_DATA", user);
      model.addAttribute("userData", user);
   }
   else {
      model.addAttribute("userData", storedUser);
   }
   return "redirect:/";
}
```

Spring Session Controller Exemplo 4/4

```
    @PostMapping("/destroy")
    public String destroySession(HttpServletRequest request) {
        request.getSession().invalidate();
        return "redirect:/";
    }
}
```

Spring Session
index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Spring Boot Session Example</title>
</head>
<body>
<div>
<h2>User Data In Session</h2>
<span th:text="${userData?.name} ?: 'No user name'" />
<br>
<span th:text="${userData?.email} ?: 'No user email'" />
<br>
<form th:action="@{/login}" method="post">
<input name="name" value="name" id="nameField">
<br> <input name="email" value="email" id="emailField">
<br> <input type="submit" value="Save User Data" />
</form>
<div>
<h2>Store Text Messages in Session</h2>
<form th:action="@{/persistMessage}" method="post">
<textarea name="msg" cols="40" rows="2"></textarea>
<br> <input type="submit" value="Save Message" />
</form>
</div>
<div>
<h2>Messages</h2>
<ul th:each="message : ${sessionMessages}">
<li th:text="${message}">msg</li>
</ul>
</div>
<div>
<h2>Session ID</h2>
Current Session ID xx <span th:text="${sessionId}" />
</div>
<div>
<form th:action="@{/destroy}" method="post">
<input type="submit" value="Destroy Session" />
</form>
</div>
</body>
</html>
```

Spring Session Index Page

- Página no navegador

Spring Session
Class User – Java Bean

```
package com.example.ufu.session;
import java.io.Serializable;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;
//Thymeleaf requer implmentar a interface
//Serializable
public class User implements Serializable{
   private Long id;
   @NotNull
   private String name;
   @Email
   private String email;
   public User() {};
   public User(String name, String email) {
      this.name = name;
      this.email = email;
   };
   public Long getId() {
     return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
     this.name = name;
   }
   public String getEmail() {
     return email;
   }
   public void setEmail(String email) {
     this.email = email;
   }
   @Override
   public String toString() {
      return "User [id=" + id + ", name=" + name    + ",
      email=" + email + ", toString()=" +
      super.toString() + "]";
   }
}
```