



Universidade do Minho
Escola de Engenharia

Trabalho Teórico 1 de Arquiteturas Aplicacionais
1º/4º ano MEI/MIEI
Frameworks Java e .NET
Relatório de Desenvolvimento

João Paulo Oliveira de Andrade Marques
(a81826@alunos.uminho.pt)

José André Martins Pereira
(a82880@alunos.uminho.pt)

Ricardo André Gomes Petronilho
(a81744@alunos.uminho.pt)

17 de Fevereiro de 2020

Conteúdo

1	Introdução	3
2	Frameworks	4
2.1	Grails	4
2.2	GWT	9
2.3	Spring	11
2.3.1	Vanatagens	11
2.3.2	Arquitetura	12
2.3.3	Exemplo prático	13
2.3.4	Análise final	15
3	Arquitetura Tipo	16
4	Conclusão	17

Lista de Figuras

2.1	Logotipo da framework Grails.	4
2.2	Criação da aplicação, e inicialização do interpretador da framework Grails	5
2.3	Estrutura do projeto criada após execução do comando <i>grails create-app nome-app</i>	5
2.4	Execução da aplicação web, através do interpretador.	5
2.5	Composição da diretoria <i>grails-app</i> , distribuindo as várias camadas, por diferentes diretorias.	6
2.6	Criação do controller denominado <i>Counter</i>	6
2.7	Classe criada automaticamente pela framework, onde é adicionado um sufixo de <i>Controller</i> e um método padrão, denominado <i>index</i>	7
2.8	Aplicação web, página inicial, com lista de controllers disponíveis.	7
2.9	Controller com mais ações definidas.	8
2.10	Execução da ação <i>index</i>	8
2.11	Execução da ação <i>increment</i>	9
2.12	Execução da ação <i>decrement</i>	9
2.13	Execução da ação <i>set</i>	9
2.14	Logotipo da framework GWT.	9
2.15	Modelo MVP	10
2.16	Código Java para <i>Quick Start</i>	10
2.17	Monitorização do servidor web, fornecida pela framework GWT , onde se pode ver os pedidos.	11
2.18	A aplicação web criada pela framework.	11
2.19	Logotipo da framework Spring.	11
2.20	Arquitetura da framework Spring.	12
2.21	Model <i>Person</i>	13
2.22	Controller <i>PersonController</i>	14
2.23	DAO <i>PersonRepository</i>	15

Capítulo 1

Introdução

Na unidade curricular de Arquiteturas Aplicacionais, foi-nos proposta a pesquisa sobre frameworks Java e .NET, que contribuem para a separação das camadas de uma aplicação.

Desta forma, no presente relatório será abordada a temática das diferentes frameworks Java para desenvolvimento de aplicações Web.

Ao longo do relatório, serão abordadas algumas frameworks, apresentando os seus pontos fortes e contributos para as arquiteturas das aplicações.

As apresentações das frameworks, são sempre acompanhadas com exemplos práticos, pois o grupo, testou as frameworks na prática, para melhor perceção.

Será também escolhida proposta uma arquitetura tipo, com a algumas das frameworks apresentadas. Na verdade, a arquitetura proposta, poderá ser utilizada no desenvolvimento do projeto prático das Unidades Curriculares do Perfil de Engenharia de Aplicações (Arquiteturas Aplicacionais e Sistemas Interativos).

Capítulo 2

Frameworks

Nesta secção serão apresentadas algumas das frameworks escolhidas pelo grupo para o trabalho a realizar.

2.1 Grails



Figura 2.1: Logotipo da framework Grails.

As frameworks web mais recentes tendem a ser bastante complicadas/complexas do que o necessário.

Desta forma, a **Grails**, acenta no conceito de framework dinâmica, tal como **Rails** e **Django**, tendo como principal objetivo, a simplicidade, reduzindo a complexidade da construção de aplicações web na plataforma Java.

A escolha desta framework para este trabalho, deveu-se ao facto da mesma centrar-se também na separação de camadas de uma aplicação web, reduzindo as preocupações do desenvolvedor.

A framework segue uma arquitetura full stack, isto é, híbrida (*server side* e *client side*), abrangindo todas as camadas de uma aplicação web. No entanto, o grupo focou-se mais nas camadas MVC (Model View and Controller), da camada de apresentação.

A *Groovy* é a linguagem de programação utilizada na framework, sendo esta orientada a objetos, desenvolvida para a plataforma *Java*, como alternativa à mesma. A linguagem caracteriza-se por ser bastante semelhante, a nível de syntax, a *Python*, *Ruby*, entre outras. Desta forma, sendo desenvolvida para a plataforma Java, a framework **Grails**, como utiliza esta linguagem, considera-se uma framework da plataforma Java, tal como os próprios desenvolvedores o afirmam.

A framework **Grails** contém um interpretador/aplicação interativo (que será mostrado de seguida em detalhe), onde se controla toda a estrutura da aplicação web, e desta forma, a framework consegue organizar o projeto, separando as várias camadas sem intervenção do desenvolvedor, de forma automática.

Com o objetivo de uma melhor perceção de como a framework **Grails** efetua a separação de camadas MVC, vai-se apresentar um exemplo prático simples, que o grupo fez para melhor entender a mesma.

diretoria, os ficheiros são divididos em diferentes sub-diretorias (camadas), de forma automática pela framework. Na figura 2.5, pode-se verificar com mais detalhe a composição desta diretoria, onde se assinala a quadrados vermelhos as camadas MVC.

2.7 Convention over Configuration

Grails uses "convention over configuration" to configure itself. This typically means that the name and location of files is used instead of explicit configuration, hence you need to familiarize yourself with the directory structure provided by Grails.

Here is a breakdown and links to the relevant sections:

- `grails-app` - top level directory for Groovy sources
 - `conf` - [Configuration sources](#)
 - `controllers` - [Web controllers](#) - The C in MVC.
 - `domain` - The [application domain](#). - The M in MVC
 - `i18n` - Support for [internationalization \(i18n\)](#).
 - `services` - The [service layer](#).
 - `taglib` - [Tag libraries](#).
 - `utils` - Grails specific utilities.
 - `views` - [Groovy Server Pages](#) or [JSON Views](#) - The V in MVC.
- `src/main/scripts` - [Code generation scripts](#).
- `src/main/groovy` - Supporting sources
- `src/test/groovy` - [Unit and integration tests](#).

Figura 2.5: Composição da diretoria *grails-app*, distribuindo as várias camadas, por diferentes diretorias.

No entanto, poderá-se pensar que o programador terá o trabalho de colocar os diferentes ficheiros, nas diferentes pastas. A resposta é "não", a própria framework tal como já foi dito, faz isso. Na verdade, para se criar um *Controller*, faz-se através do interpretador (figura 2.6), e desta forma, a framework consegue controlar e organizar as diferentes camadas.

Após a criação do controlador, é criada a classe do mesmo, na diretoria *controllers*, com o sufixo *Controller* ao nome dado na criação. A classe gerada, contém um método padrão, denominado *index*, que corresponde a uma ação (figura 2.7), sendo que um controlador pode ter várias ações, ou seja, vários métodos.

```
grails> create-controller counter
| Created grails-app/controllers/app/CounterController.groovy
| Created src/test/groovy/app/CounterControllerSpec.groovy
grails> 
```

Figura 2.6: Criação do controller denominado *Counter*.

```

CounterController.groovy x
1 package app
2
3 class CounterController {
4
5     def index() { }
6 }
7

```

Figura 2.7: Classe criada automaticamente pela framework, onde é adicionado um sufixo de *Controller* e um método padrão, denominado *index*.

A seguir, à criação do controlador, o mesmo já fica visível na homepage da aplicação web (figura 2.8). A forma de aceder ao controlador e às respetivas ações, segue o seguinte padrão `http://localhost:8080/controller-name/action` (no caso de argumentos `http://localhost:8080/controller-name/action?agr1=value1`).

Daqui advém, os pontos fortes desta framework, para além da separação das camadas MVC, vistas anteriormente, a redução da complexidade, pois compara-se o nome do URL, com o nome do método, sabendo o método a executar, não sendo necessárias anotações ou outras alternativas, presentes em outras frameworks. Do mesmo modo, pedidos do tipo **GET** com argumentos, também se tornam simples, pois basta o método receber os argumentos com o mesmo nome dado no URL, tal como se pode observar na figura 2.9 e 2.13 da ação *set*.

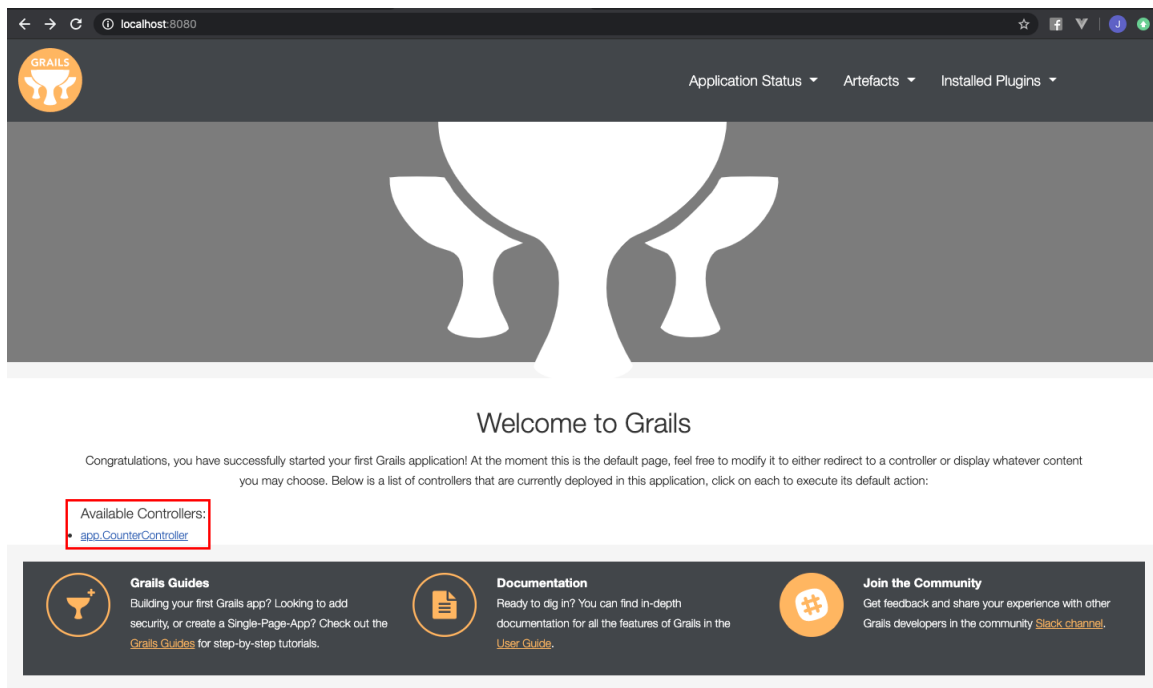


Figura 2.8: Aplicação web, página inicial, com lista de controllers disponíveis.

Assim, após a perceção do funcionamento dos controllers, desenvolvemos várias ações para o mesmo controller, tal como se pode verificar, na figura 2.9.

Importante realçar que o método *render*, envia as strings ou valores para uma view criada automatica-

mente, caso esta ainda não exista, ou seja, a framework abstrai o desenvolvedor desse processo, tornando-o simples. De seguida, apresentamos os vários outputs para as diversas ações definidas, nas figuras, 2.10, 2.11, 2.12 e 2.13.

```
CounterController.groovy x
package app

class CounterController {

    int value = 0

    /* standard method */
    def index() {
        render "Standard method!"
    }

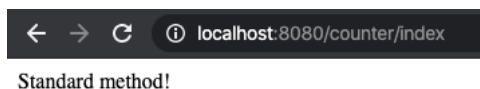
    /* increment value */
    def increment(){
        this.value++
        display()
    }

    /* decrement value */
    def decrement(){
        this.value--
        display()
    }

    /* display value */
    def display(){
        render "value = " + this.value
    }

    /* set value */
    def set(int value){
        this.value = value
        display()
    }
}
```

Figura 2.9: Controller com mais ações definidas.

A screenshot of a web browser window. The address bar shows the URL 'localhost:8080/counter/index'. Below the address bar, the text 'Standard method!' is displayed on the page.

← → ↻ ⓘ localhost:8080/counter/index
Standard method!

Figura 2.10: Execução da ação index.

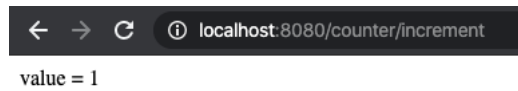


Figura 2.11: Execução da ação increment.

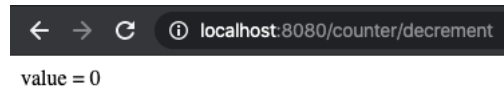


Figura 2.12: Execução da ação decrement.

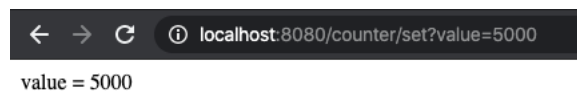


Figura 2.13: Execução da ação set.

Assim, conclui-se em relação à framework **Grails**, que faz uma boa divisão das camadas MVC, bem como de outros níveis da aplicação, tornando-as independentes. Também importante realçar a característica de redução de complexidade e simplicidade desta framework.

2.2 GWT



Figura 2.14: Logotipo da framework GWT.

A **GWT** consiste numa framework para construir e otimizar aplicações web complexas.

Esta framework foi criada pela Google com o intuito de ser simples para os programadores desenvolverem a aplicação web e ser muito eficiente para os utilizadores. É uma framework client-side, apesar de ter um servidor a correr, este apenas serve para disponibilizar a view. Além disso esta framework não segue o modelo **MVC**, mas sim o modelo **MVP** (Model-View-Presenter).

O **MVP** consiste em o utilizador interagir com a View, esta notifica o Presenter que por sua vez atualiza o Model, de seguida o Presenter obtém dados do Model e encaminha-os para a View para esta ser atualizada. A seguinte figura ilustra o funcionamento do modelo **MVP**.

Model View Presenter



Figura 2.15: Modelo MVP.

Algumas das vantagens de utilizar o **GWT**:

- Fornece API's Java e Widgets que permitem a escrita de aplicações AJAX em Java que depois são compiladas para JavaScript;
- Permite a colocação de "split-points" no código o que fará com que aplicações muito grandes possam ser divididas em vários segmentos JavaScript para um arranque mais rápido da página Web enquanto descarrega o resto dos fragmentos;
- Possui duas ferramentas de otimização para que o JavaScript gerado através do Java seja o mais eficiente possível, mas que também evite incompatibilidades entre diversos browsers;
- Os erros são encontrados em tempo de compilação;
- Pode ainda ser utilizado JavaScript entre o código Java através do JSNI (JavaScript Native Interface).

De seguida será ilustrado um pequeno exemplo para mostrar a simplicidade da framework.

```
MyWebApp.java
1 package com.mycompany.mywebapp.client;
2
3 import com.google.gwt.core.client.EntryPoint;
4 import com.google.gwt.user.client.ui.Label;
5 import com.google.gwt.user.client.ui.RootPanel;
6
7 public class MyWebApp implements EntryPoint {
8     @Override
9     public void onModuleLoad() {
10         final Label label = new Label("Hello World!");
11         RootPanel.get().add(label);
12     }
13 }
```

Figura 2.16: Código Java para *Quick Start*.

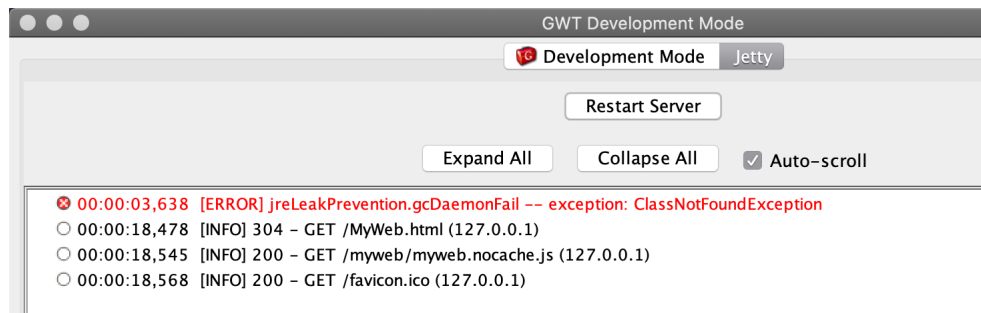
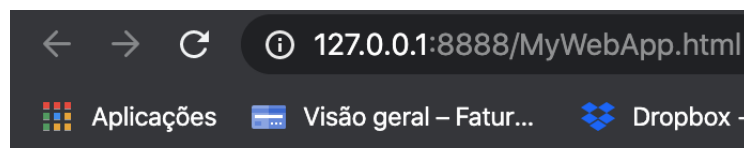


Figura 2.17: Monitorização do servidor web, fornecida pela framework **GWT**, onde se pode ver os pedidos.



Hello World!

Figura 2.18: A aplicação web criada pela framework.

2.3 Spring



Figura 2.19: Logotipo da framework Spring.

A **Spring** é uma framework desenvolvida pela equipa **pivotal** usando Java dedicada principalmente à implementação de lógica aplicacional back-end.

2.3.1 Vanatagens

Existem enúmeras vantagens ao utilizar esta framework tais como:

- é **escrita em Java**, beneficiando do facto de ser uma linguagem muito usada e preparada para projetos de elevada complexidade (usando conceitos de POO). Outra vantagem da utilização de Java é a comunidade (tanto documentação oficial como tutoriais e explicações de terceiros) ser grande e consistente, tornando o suporte a dúvidas e obstáculos do desenvolvimento mais facilitados de se superar.
- diversos desenvolvedores fidedignos, tais como: Alibaba, Amazon, Google, Microsoft, entre outros; **confiam na framework**, sendo um indicador positivo a favor da utilização da mesma.
- é uma framework **flexível** uma vez que a sua arquitetura base utiliza os padrões **Inversion of Control (IoC)** e **Dependency Injection (DI)** tornando a implementação do código facilitada uma vez que parte considerável da lógica é automaticamente introduzida.

- contém inúmeras **bibliotecas de terceiros** integradas na arquitetura base constituindo um conjunto de ferramentas disponível para qualquer tipo de aplicação tanto server-side como client-side. Por exemplo a framework [hibernate](#) é facilmente integrada no modelo de dados de uma API RESTful já escrita em Spring, tornando o processo de persistência de dados intuitivo e praticamente automático.
- contém o **plugin Spring boot** que automatiza a criação da infraestrutura inicial do projeto: inicializando e configurando o servidor aplicacional [Tomcat](#), verifica e descarrega bibliotecas necessárias para o contexto da aplicação e disponibiliza métricas para avaliar o desempenho e estado do sistema, podendo ser remotamente avaliado.
- é uma framework **segura**. Utilizar bibliotecas de terceiros é sempre um risco uma vez que os seus desenvolvedores podem simplesmente desistir e descontinuar atualizações das bibliotecas ficando as mesmas sem suporte e sujeitas a vulnerabilidades ou erros. Desta forma a equipa da Spring monitoriza e verifica o estado das bibliotecas terceiras integradas na sua arquitetura certificando que o seu uso é seguro.
- por último mas não menos importante a Spring, tal como Java, tem uma **comunidade grande e prestável** com inúmeros tutoriais e artigos na própria documentação oficial.

2.3.2 Arquitetura

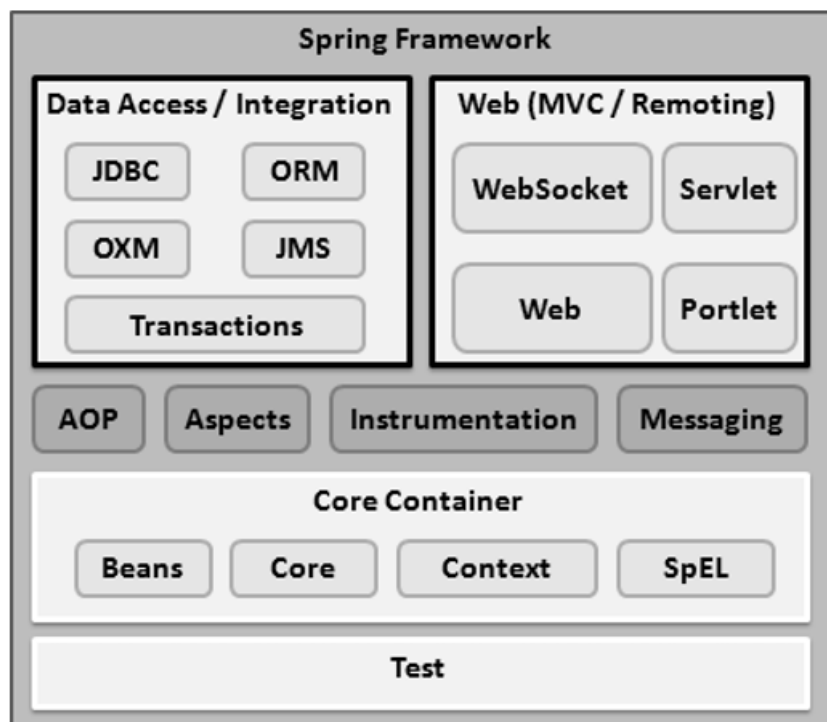


Figura 2.20: Arquitetura da framework Spring.

Core Container

O Core Container contém as principais classes que formam a arquitetura base da framework.

- O módulo Core contém a implementação dos padrões arquiteturais: IoC e DI; referidos anteriormente.

- O módulo Bean implementa o padrão Factory (usado para criação genérica de objetos), neste caso é uma fábrica de Beans.
- O módulo Context utiliza os módulos Core e Bean para gerar automaticamente os objetos configurados pela implementação específica do programador.
- O módulo SpEL é utilizado pela Spring para gerar gráficos e monitorizar os objetos gerados e configurados pela própria framework.

Outros Containers

O container Data Access implementa as classes necessárias ao acesso e manipulação de dados nas diferentes base de dados existentes.

O container Web implementa o padrão MVC utilizado para responder a pedidos HTTP ou outros protocolos.

2.3.3 Exemplo prático

Nesta secção segue-se um exemplo prático no qual foi criada uma API RESTful expondo a lógica back end através de métodos HTTP.

Model - classe Person

Inicialmente definiu-se uma classe Person que representa a informação de uma Pessoa, contendo o seu id, nome, idade e email. A vermelho em cima da declaração de cada variável de instância é possível observar anotações.

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private String id;

    @Min(8)
    @Max(64)
    @NotNull
    private String name;

    @Positive
    @NotNull
    private int age;

    @Email
    @NotEmpty
    private String email;
```

Figura 2.21: Model Person.

- @GeneratedValue(...) - indica que a variável id é gerada automaticamente pela framework, neste caso é uma sequência de inteiros começado em 1 em formato String.
- @Min(8) - indica que a string (nome) tem de ter no mínimo 8 caracteres.
- @Max(64) - indica que a string (nome) tem de ter no máximo 64 caracteres.
- @NotNull - indica que a variável tem de ter necessariamente um valor.
- @NotEmpty - indica que a variável tem de ter necessariamente um valor não vazio.
- @Positive - indica que o inteiro (idade) tem de ser positivo.
- @Email - indica que a string tem de ter um formato válido de email.

Existem outras anotações o importante é que a Spring juntamente com a framework hibernate integrada, valida automaticamente os valores atribuídos no momento de criação do objeto Person, lançando uma exceção quando os mesmos são inválidos.

A anotação **@Entity** indica á hibernate que a classe refere-se a uma tabela na base de dados. A anotação **@Id** indica que a variável id é chave primária nessa mesma tabela.

Controller - classe PersonController

Tendo definido o modelo de dados (classe Person) é necessário definir a classe que implementa os métodos que processam, neste caso, os pedidos HTTP associados a este objeto, a classe PersonController.

```
@RestController
@RequestMapping(path="/person")
public class PersonController {

    @Autowired
    public PersonRepository personRepository;

    @GetMapping("/all")
    public ResponseEntity<Iterable<Person>> getPerson() {
        return new ResponseEntity<>(personRepository.findAll(), HttpStatus.OK);
    }

    @GetMapping("/get")
    public ResponseEntity<Person> getPerson(@RequestParam(value = "id", defaultValue = "") String id) {
        if (personRepository.existsById(id) == false) return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else return new ResponseEntity<>(personRepository.findById(id).get(), HttpStatus.OK);
    }

    @PostMapping("/put")
    public ResponseEntity<Person> putPerson(@Valid @RequestBody Person person) {
        if (personRepository.existsById(person.getName())) return new ResponseEntity<>(HttpStatus.CONFLICT);
        else {
            personRepository.save(person);
            return new ResponseEntity<>(HttpStatus.CREATED);
        }
    }
}
```

Figura 2.22: Controller PersonController.

- @RestController indica à Spring que esta classe é um Controller e por isso de acordo com o mapeamento das rotas pode processar HTTP Requests.
- RequestMapping(...) indica o nome da rota (no url) para evocar este controlador.
- @GetMapping(...) indica que este método processa HTTP GETs evocados pela rota definida no url.
- @PostMapping(...) indica que este método processa HTTP POSTs evocados pela rota definida no url.
- @Valid tal como referido anteriormente é utilizado para validar os dados introduzidos na classe model (neste caso Person).
- @RequestBody indica que o objeto em causa foi exportado do payload (body) do pedido HTTP.
- @RequestParam indica que a variável em causa corresponde a um parâmetro específico do pedido HTTP.
- @Autowired é uma das anotações mais importante e eficaz, indica à framework hibernate que o objeto (neste caso PersonRepository) que é responsável pela persistência dos dados (DAO) é automaticamente gerado e criado (criação dinâmica do objeto), sem qualquer intervenção do programador.

DAO - interface PersonRepository

Esta interface provavelmente é a que demonstra de forma mais óbvia o poder de utilizar uma framework no sentido da **simplicidade e automação** de desenvolvimento de código.

```
@Repository
public interface PersonRepository extends CrudRepository<Person, String> {}
```

Figura 2.23: DAO PersonRepository.

Tal como se observa na figura, apenas indicando que a interface é do tipo CrudRepository e refere-se ao modelo de dados Person, a Spring juntamente com o Hibernate gera e cria dinamicamente uma classe que implementa esta interface fazendo a ligação entre os modelos e a sua preexistência na base de dados.

2.3.4 Análise final

Em suma, a criação das respetivas classes: Model, Controller e DAO; tornam-se simples e intuitiva uma vez que a Spring juntamente com as bibliotecas de terceiros integradas automatizam e gerem dinamicamente classes e dependências necessárias.

Capítulo 3

Arquitetura Tipo

Com uma análise detalhada, foram escolhidas as frameworks **Grails** e **Spring** para a proposta de arquitetura tipo. A framework **Grails** será utilizada para o frontend (client-side) da aplicação, por outro lado, a framework **Spring** será utilizada para o backend (server-side).

Na verdade, se na framework **GWT** fossem acrescentados outros packages (tal como no Spring foi adicionado o **Spring Boot** para melhorar a framework), seria possível ter o client-side e o server-side com a **GWT** e através do **Hibernate** realizar o acesso à base de dados. No entanto, a framework Spring é bastante mais simples que a **GWT** com os packages extra. Desta forma, fica mais fácil integrar a mesma com o Hibernate, ficando um backend completo e simples.



FrontEnd



BackEnd



DataBase

Capítulo 4

Conclusão

Em suma, após a conclusão deste trabalho, percebemos a grande variedade de frameworks Java Web existentes. Desta forma, escolheu-se para análise, aquelas que cumprem os requisitos do enunciado, sendo também as que nos pareceram menos complexas de analisar, contendo melhor documentação, bem como um boa comunidade para ajuda.

Algumas frameworks que analisamos, mas que não estão presentes no relatório, tais como **Blade**, **Struts**, entre outras, deve-se ao facto de serem complexas, ou de não conterem uma boa documentação, sendo difícil a sua análise. No entanto, com mais tempo, provavelmente o grupo irá, experimentá-las.

Do mesmo modo, o grupo decidiu não analisar frameworks .NET, pois, apesar de já ter sido usado por exemplo ASP.NET em Laboratórios de Informática 4, achou-se mais interessante as frameworks de Java, visto a maior familiarização com a linguagem.