

INE5646 Programação para Web

- Tópico :
Aplicações Baseadas na Java Virtual Machine
(JVM)
- Subtópico:
Play (Parte 2)

(estes slides fazem parte do material didático da disciplina
INE5646 Programação para Web)

Sumário

- **Parte 1:**
 - Instalação, configuração e administração de servidores.
 - Processamento síncrono e assíncrono de requisições
 - Template engine.
- **Parte 2:**
 - Acesso a bases de dados.
 - Serviços web.
 - Processamento de stream de dados.

Acesso a Bases de Dados

- Aplicações Play acessam bases de dados por meio do *driver JDBC* fornecido pelo fabricante do banco.
- Desenvolver uma aplicação para web em Play que acesse uma base de dados implica nas seguintes providências:
 - Copiar o driver JDBC do banco para dentro do diretório **lib** da aplicação.
 - Definir, no arquivo **conf/application.conf**, as informações típicas de acesso (qual base de dados será acessada, nome e senha do usuário com permissão de acesso à base de dados) mais as informações relativas ao pool de conexões.
 - Se a base de dados não existir, definir em SQL a estrutura (tabelas, relacionamentos, índices, etc) da base de dados usando uma estratégia chamada “evolutions”.
 - Implementar o código (Java ou Scala) que acessa o banco. Pode-se usar diretamente a API JDBC, algum framework Java (como o Hibernate) ou framework Scala (atualmente o Anorm, já incluso no Play, ou o framework Slick <http://slick.typesafe.com/>)

Aplicação Exemplo

- Uma aplicação exemplo, mostrada a seguir, é usada para demonstrar como as providências citadas são implementadas.
- A aplicação consiste em um sistema de registro de ocorrências. Exemplo: “O usuário Fulano de tal relata que no dia 15 de fevereiro de 2013 viu um meteoro cair na cidade russa de Chelyabinsk”.
- As funcionalidades disponíveis são:
 - Cadastrar usuário (nome e código)
 - Cadastrar ocorrência (usuário, data (opcional) e descrição da ocorrência)
 - Listar todos os usuários cadastrados
 - Listar todas as ocorrências cadastradas
 - Listar todas as ocorrências de um determinado usuário

Página inicial

The screenshot shows a web browser window with the address bar set to 'localhost:9000'. The page title is 'INE5646 - demoBD'. Below the title, the main heading is 'Usuários e Ocorrências'. The page is divided into two sections: 'Usuários' and 'Ocorrências'. Each section contains a list of actions with associated form fields and buttons.

INE5646 - demoBD

Usuários e Ocorrências

Usuários

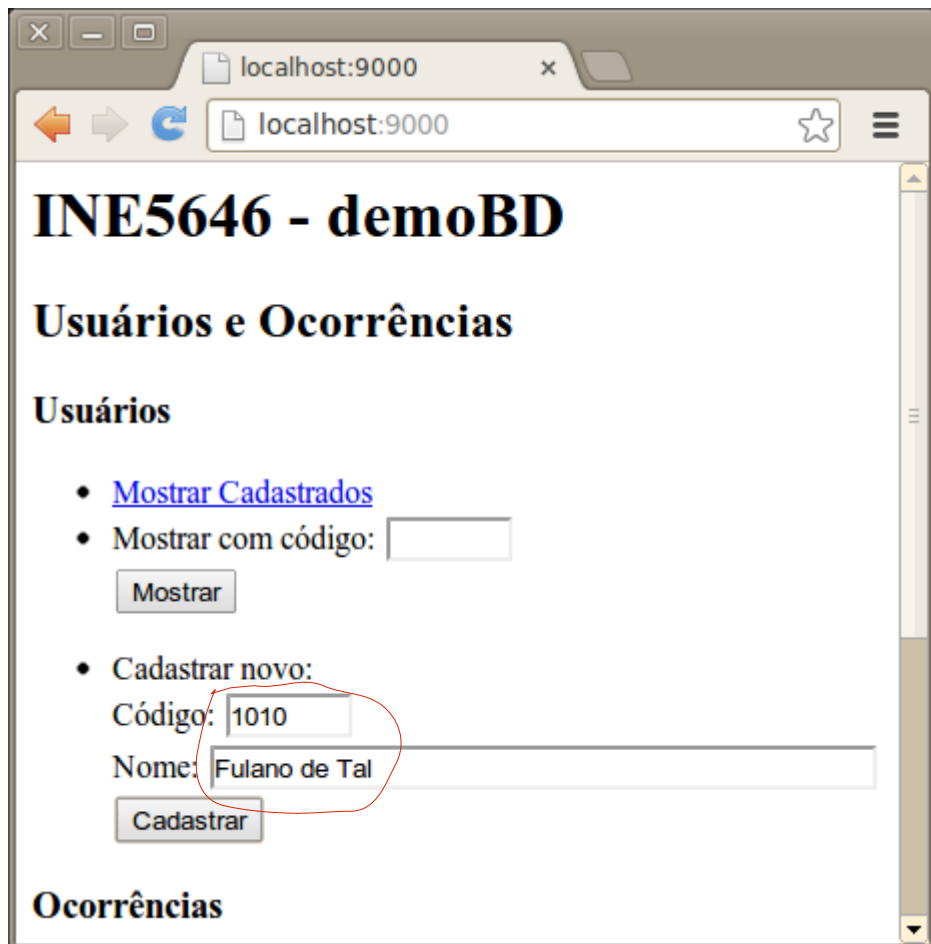
- [Mostrar Cadastrados](#)
- Mostrar com código:
- Cadastrar novo:
Código:
Nome:

Ocorrências

- [Mostrar Cadastradas](#)
- Mostrar associadas ao Usuário código:
- Cadastrar nova:
Código do Usuário:
Data :
Descrição:

Pág. inicial – Cadastrando Usuário

- Cadastrando usuário 1010 – Fulano de Tal



localhost:9000

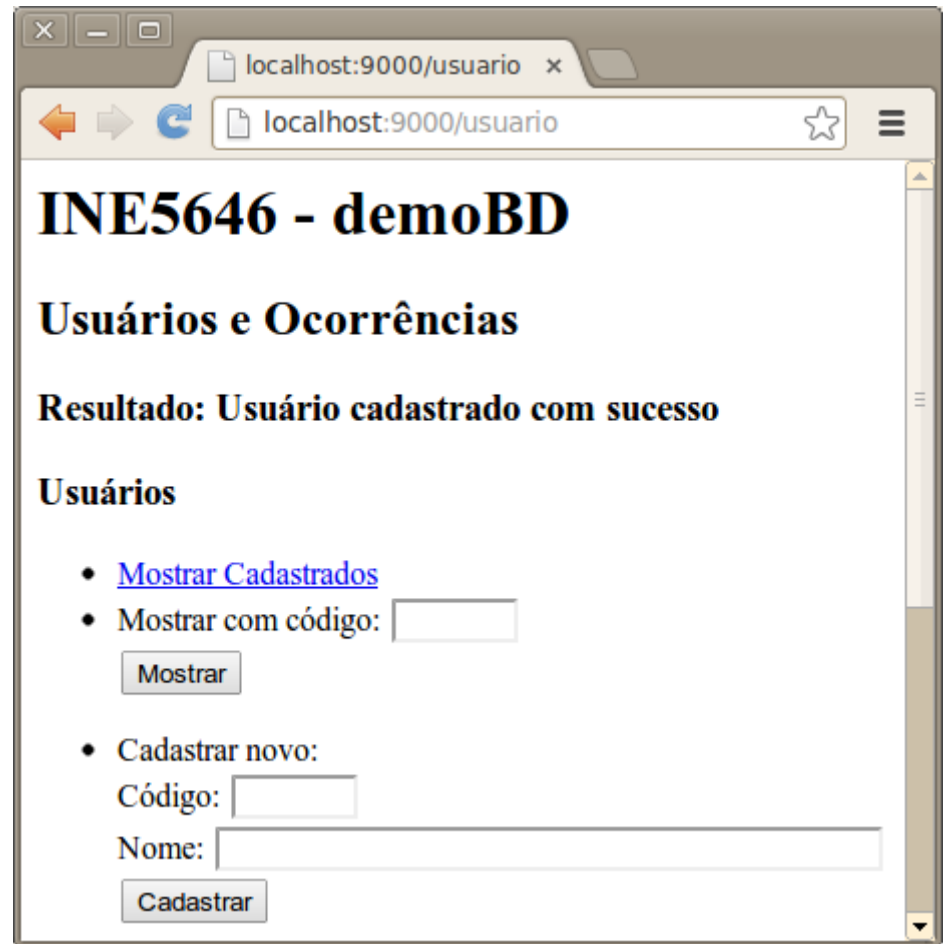
INE5646 - demoBD

Usuários e Ocorrências

Usuários

- [Mostrar Cadastrados](#)
- Mostrar com código:
- Cadastrar novo:
Código:
Nome:

Ocorrências



localhost:9000/usuario

INE5646 - demoBD

Usuários e Ocorrências

Resultado: Usuário cadastrado com sucesso

Usuários

- [Mostrar Cadastrados](#)
- Mostrar com código:
- Cadastrar novo:
Código:
Nome:

Página Usuários Cadastrados

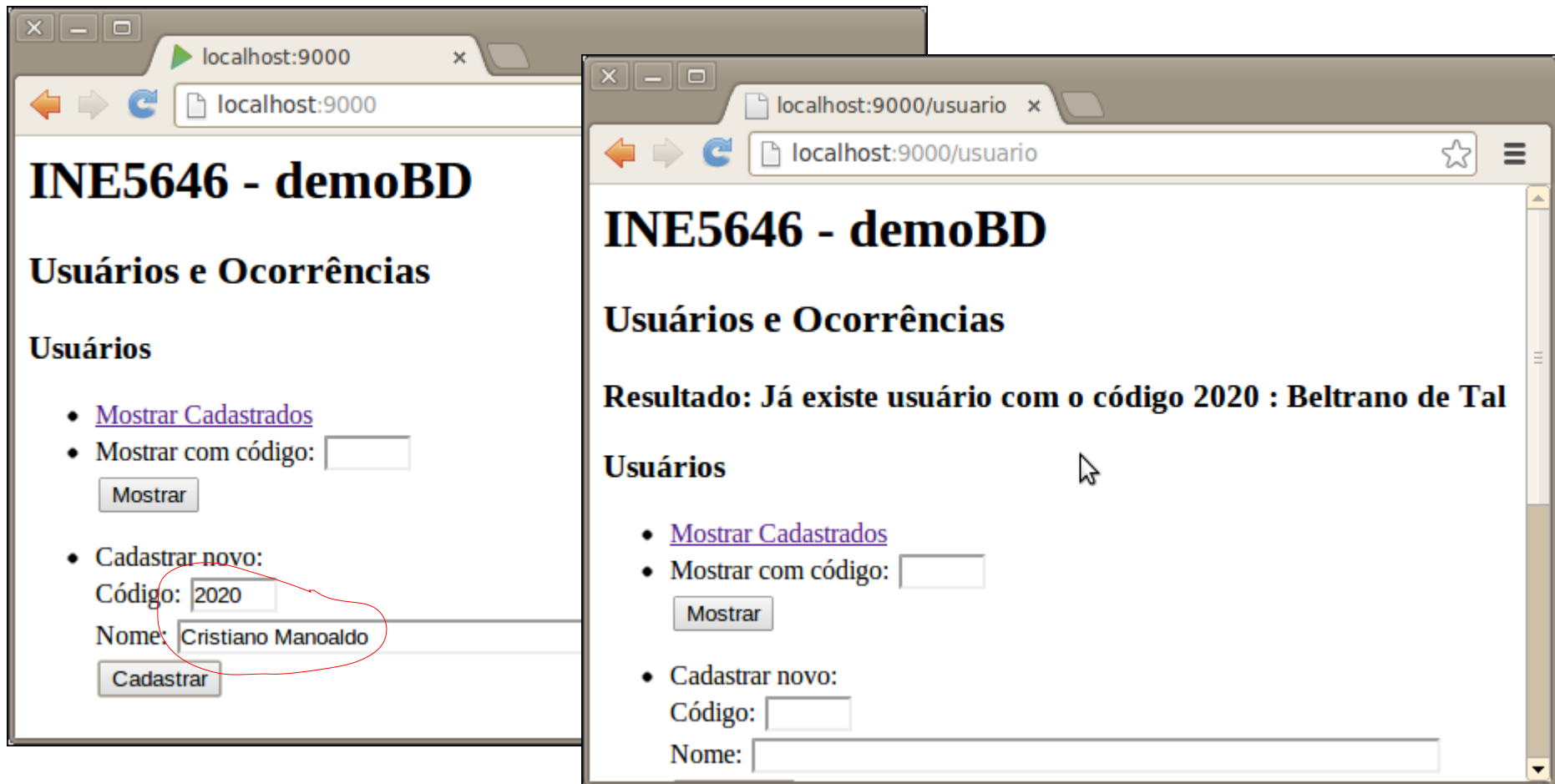
- Mostrando os usuários cadastrados:



Código	Nome
2020	Beltrano de Tal
1010	Fulano de Tal
3030	Geremias Come Folha

Pág. Inicial – Cadastro de Usuário

- Tentativa de cadastrar usuário já cadastrado (mesmo código):



Pág. Inic. – Cadastro de Ocorrência

- Cadastrando uma ocorrência para usuário 2020:

localhost:9000

Código:

Nome:

Cadastrar

Ocorrências

- [Mostrar Cadastradas](#)
- Mostrar associadas ao Usuário código:
Mostrar
- Cadastrar nova:
Código do Usuário:
Data :
Descrição:
Cadastrar

localhost:9000/ocorrencia

INE5646 - demoBD

Usoários e Ocorrências

Resultado: Ocorrencia cadastrada com sucesso!

Usoários

- [Mostrar Cadastrados](#)
- Mostrar com código:
Mostrar
- Cadastrar novo:
Código:
Nome:

Pág. Inic. – Cadastro de Ocorrência

- Tentativa de cadastrar ocorrência para usuário não cadastrado:

localhost:9000/ocorren x
localhost:9000/ocorren

Código:

Nome:

Cadastrar

Ocorrências

- [Mostrar Cadastradas](#)
- Mostrar associadas ao Usuário código:
Mostrar
- Cadastrar nova:
Código do Usuário:
Data :
Descrição:
Cadastrar

localhost:9000/ocorren x
localhost:9000/ocorren

INE5646 - demoBD

Usuários e Ocorrências

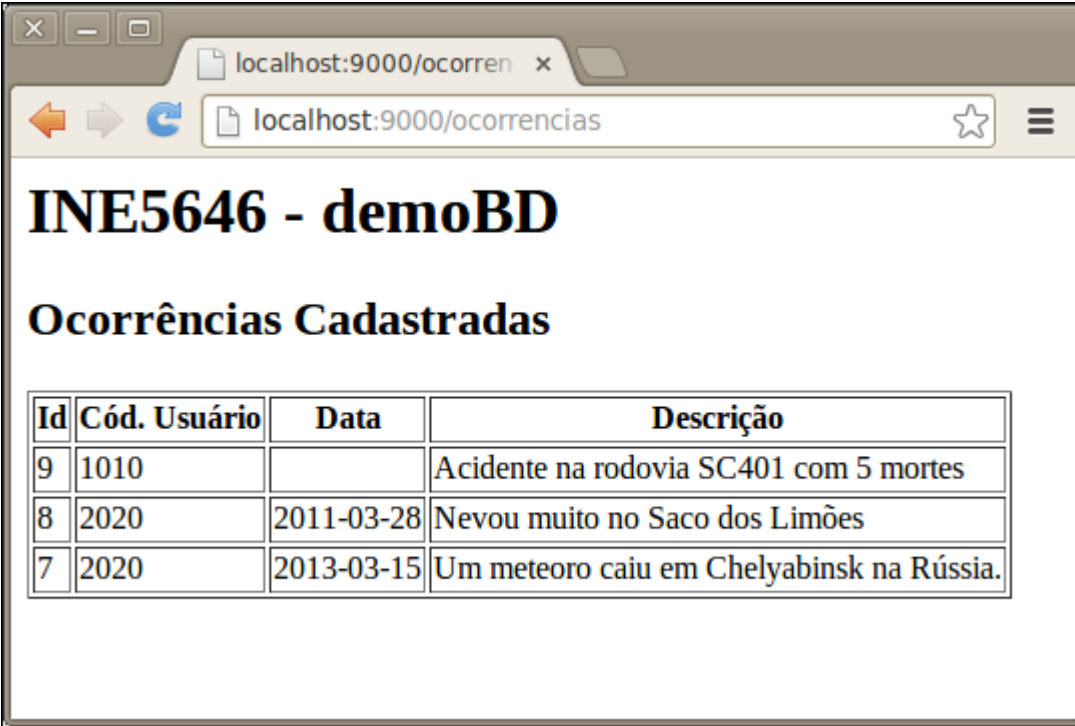
Resultado: Não existe usuário com o código 9090

Usuários

- [Mostrar Cadastrados](#)
- Mostrar com código:
Mostrar
- Cadastrar novo:
Código:
Nome:

Página Ocorrências

- Mostrando ocorrências cadastradas:



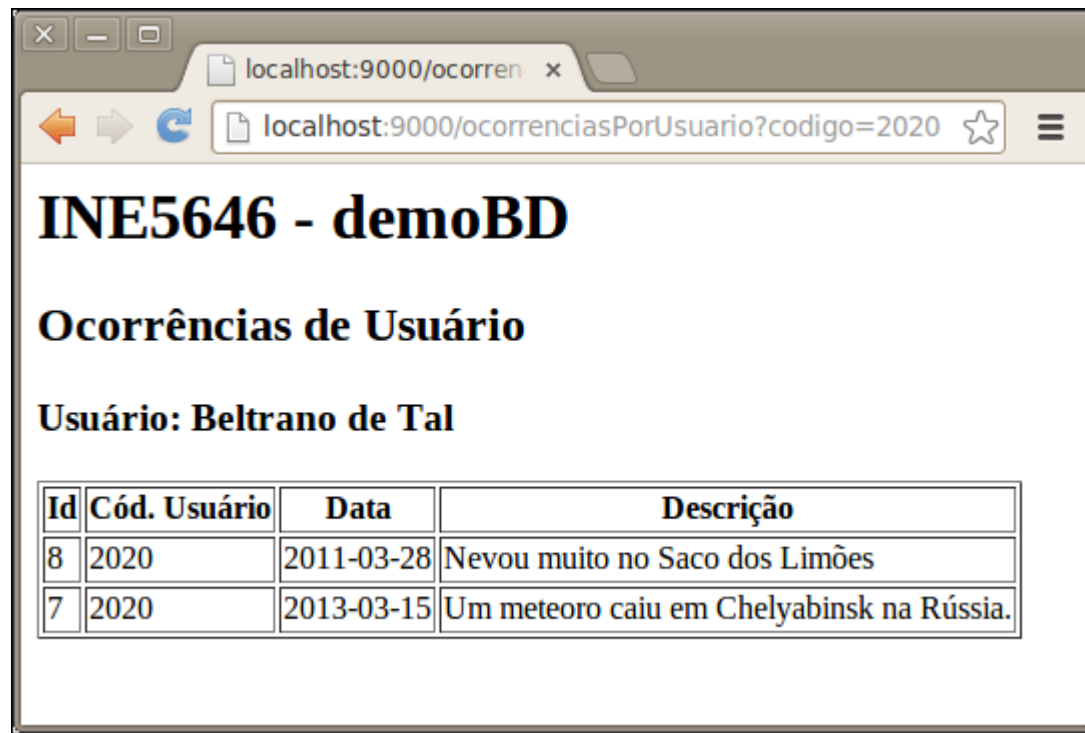
INE5646 - demoBD

Ocorrências Cadastradas

Id	Cód. Usuário	Data	Descrição
9	1010		Acidente na rodovia SC401 com 5 mortes
8	2020	2011-03-28	Novou muito no Saco dos Limões
7	2020	2013-03-15	Um meteoro caiu em Chelyabinsk na Rússia.

Página Ocorrências de Usuário

- Mostrando ocorrências do usuário 2020:



INE5646 - demoBD

Ocorrências de Usuário

Usuário: Beltrano de Tal

Id	Cód. Usuário	Data	Descrição
8	2020	2011-03-28	Novou muito no Saco dos Limões
7	2020	2013-03-15	Um meteoro caiu em Chelyabinsk na Rússia.

Página Ocorrências de Usuário

- Mostrando que o usuário 3030 não possui ocorrências registradas:



Página Ocorrências de Usuário

- Tentativa de mostrar ocorrências para usuário não cadastrado:



Definição do Banco de Dados

- Trecho do arquivo **conf/application.conf** que define que a aplicação:
 - usa MySQL (linha 42)
 - usa a base de dados demoBDPlay que está armazenada no mesmo computador (localhost) da aplicação (linha 43)
 - Conecta-se ao banco por meio do usuário “demoBDUsuario” (linha 44) e senha “demoBDSenha” (linha 45)

```
42 db.demoBD.driver=com.mysql.jdbc.Driver
43 db.demoBD.url="jdbc:mysql://localhost/demoBDPlay"
44 db.demoBD.user="demoBDUsuario"
45 db.demoBD.password="demoBDSenha"
```

Definição do Banco de Dados

- Trecho do arquivo **conf/application.conf** também identifica que na aplicação o nome (alias) da base de dados será “**demoBD**”:

```
42 db.demoBD.driver=com.mysql.jdbc.Driver
43 db.demoBD.url="jdbc:mysql://localhost/demoBDPlay"
44 db.demoBD.user="demoBDUsuario"
45 db.demoBD.password="demoBDSenha"
```

- Outros itens também poderiam ser configurados: tamanho do pool de conexões, número de conexões, etc.

Evolutions

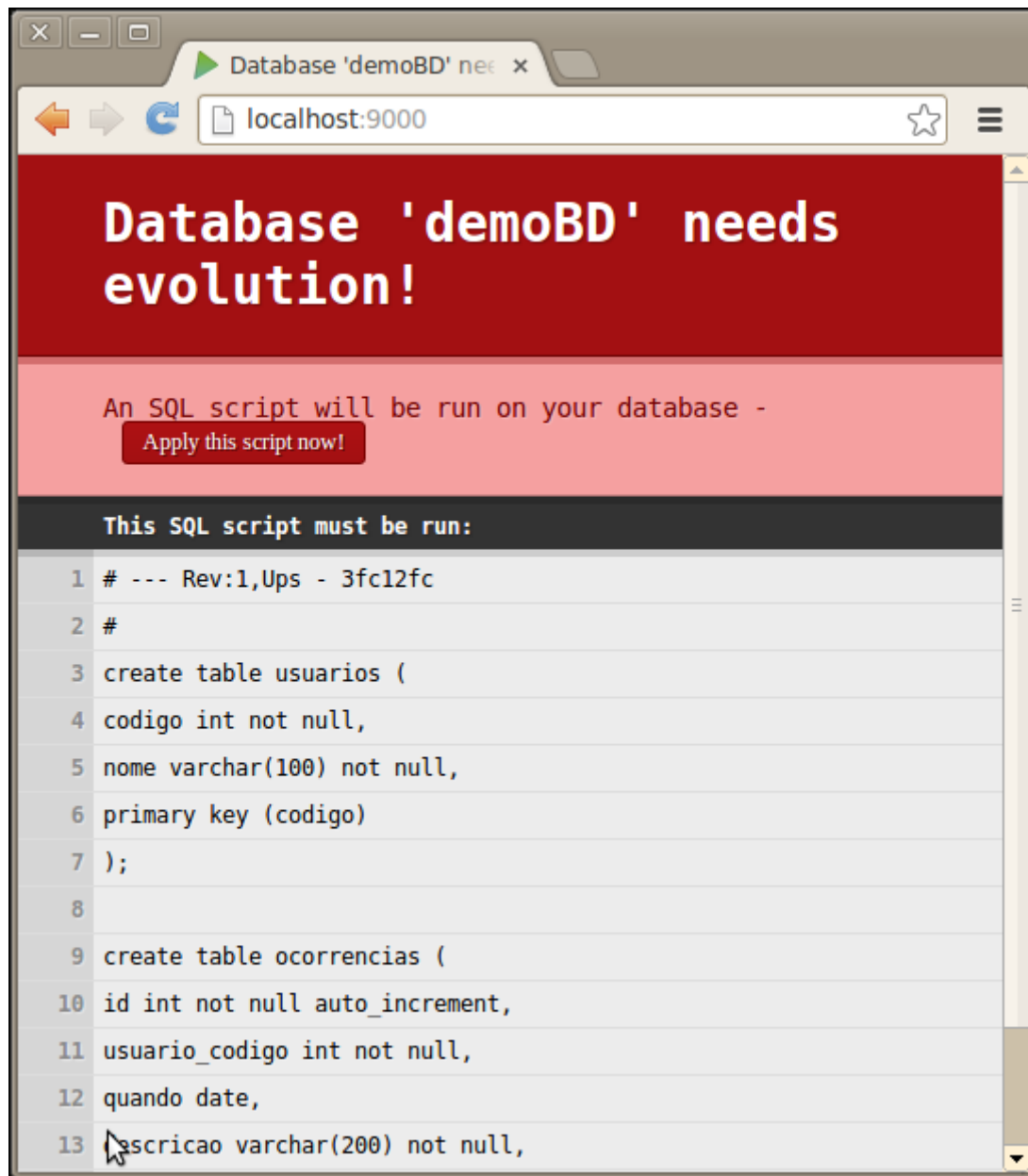
- O Play utiliza uma tecnologia chamada **evolutions**:
 - O desenvolvedor define/modifica um arquivo SQL necessário para criar e destruir as tabelas usadas pela aplicação.
 - Durante o desenvolvimento, sempre que o servidor perceber que a estrutura da base de dados é diferente da descrição contida no arquivo SQL então ele usa o arquivo para recriar (apagando os dados!) as tabelas da base de dados.

Evolutions

- O arquivo conf/evolutions/demoBD/1.sql:

```
1.sql x
1 # INE5646 - demoBD
2 # --- !Ups
3 #
4 create table usuarios (
5     codigo int not null,
6     nome varchar(100) not null,
7     primary key (codigo)
8 );
9
10 create table ocorrencias (
11     id int not null auto_increment,
12     usuario_codigo int not null,
13     quando date,
14     descricao varchar(200) not null,
15     primary key (id),
16     foreign key (usuario_codigo) references usuarios(codigo)
17 );
18
19 # --- !Downs
20
21 drop table ocorrencias;
22 drop table usuarios;
```

Evolutions



- Página mostrando o script SQL que será executado para criar as tabelas da base de dados usada pela aplicação.
- O desenvolvedor então clica sobre o botão “Apply this script now!” e o script SQL é executado.

MVC – Arquivo conf/routes

- Define como o controlador (objeto Application) deve tratar as requisições.

routes		
1	#Método URL	Ação
2	#-----	----
3		
4	GET /	controllers.Application.index
5		
6	GET /usuarios	controllers.Application.pesquiseTodosUsuarios
7		
8	GET /usuarioPorCodigo	controllers.Application.pesquiseUsuarioPorCodigo
9		
10	POST /usuario	controllers.Application.cadastreUsuario
11		
12	GET /ocorrencias	controllers.Application.pesquiseTodasOcorrencias
13		
14	GET /ocorrenciasPorUsuario	controllers.Application.pesquiseOcorrenciasPorUsuario
15		
16	POST /ocorrencia	controllers.Application.cadastreOcorrencia

MVC – Arquivo Application.scala

```
Application.scala x
1 package controllers
2
3 import play.api.mvc.{Controller, Action}
4 import play.api.data.{Form}
5 import play.api.data.Forms.{tuple,number,text,optional,date}
6
7 import models.CRUD
8 import models.dados.{Usuario, Ocorrencia}
9
10 object Application extends Controller {
11   val crud = new CRUD
12
13   def index = Action {
14   }
15
16   // usuários
17
18   def cadastreUsuario = Action { implicit request =>
19   }
20
21   def pesquiseTodosUsuarios = Action {
22   }
23
24   def pesquiseUsuarioPorCodigo = Action { implicit request =>
25   }
26
27   // ocorrencias
28
29   def pesquiseTodasOcorrencias = Action {
30   }
31
32   def pesquiseOcorrenciasPorUsuario = Action { implicit request =>
33   }
34
35   def cadastreOcorrencia = Action { implicit request =>
36   }
37 }
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
```

- O objeto Application representa o controlador:
 - Define como tratar cada requisição enviada pelo browser.
 - Define qual resposta (página HTML) deve ser enviada ao browser.

MVC – Controlador

- Application é um **objeto** e não uma classe (padrão de projeto singleton) pois só existe uma única instância para atender a todas as requisições.
- Obs: poderia haver mais de um controlador.

```
1 package controllers
2
3 import play.api.mvc.{Controller, Action}
4 import play.api.data.{Form}
5 import play.api.data.Forms.{tuple,number,text,optional,date}
6
7 import models.CRUD
8 import models.dados.{Usuario, Ocorrencia}
9
10 object Application extends Controller {
11     val crud = new CRUD
12 }
```

MVC – Controlador

- O controlador conta com a ajuda do objeto **crud** (da classe CRUD) (linha 11).
- Observar que CRUD faz parte do modelo (pacote models) e, portanto, não deve saber nada sobre o protocolo HTTP.

```
1 package controllers
2
3 import play.api.mvc.{Controller, Action}
4 import play.api.data.{Form}
5 import play.api.data.Forms.{tuple,number,text,optional,date}
6
7 import models.CRUD
8 import models.dados.{Usuario, Ocorrencia}
9
10 object Application extends Controller {
11     val crud = new CRUD
12 }
```


MVC – Controlador

- O arquivo **conf/routes** diz: requisições HTTP do tipo get para a URL “/” devem ser tratadas pelo método **index** do objeto Application.
- O objeto Application diz: responder ao browser enviando a página HTML gerada a partir da execução da função **views.html.index** (linha 14).

```
1 #Método URL      Ação
2 #-----
3
4 GET      /      controllers.Application.index
```

```
13 def index = Action {
14   Ok(views.html.index())
15 }
16
```


MVC – Cadastrando usuário

- Para cadastrar um usuário, o controlador:
 - Obtém o código e o nome do usuário (linha 21) contidos na requisição HTTP. Estes dados estavam no corpo da requisição via formulário com os campos “codigo” e “nome” (linha 20).
 - O objeto **crud** faz o cadastramento e retorna uma resposta (linha 22).
 - A página **index**, com a resposta, é montada e enviada como resposta ao browser.

```
17 // usuários
18
19 def cadastreUsuario = Action { implicit request =>
20   val form = Form(tuple("codigo" -> number, "nome" -> text))
21   val (codigo, nome) = form.bindFromRequest.get
22   val resposta = crud cadastreUsuario(Usuario(codigo, nome))
23
24   Ok(views.html.index(Some(resposta)))
25 }
26
```

MVC – Pesquisando todos usuários

- Quando for solicitado ao controlador para que ele retorne uma página contendo todos os usuários cadastrados, ele:
 - Retornará a página resultante da execução da função **views.html.usuarios** (linha 28).
 - Esta função requer um parâmetro que é uma lista contendo objetos da classe **Usuario**. Esta lista é obtida por meio do objeto **crud**.
- Observe que o compilador Scala infere todos os tipos dos dados envolvidos no método. Em Java, teríamos que digitar todos os tipos.

```
27     def pesquiseTodosUsuarios = Action {  
28         Ok(views.html.usuarios(crud pesquiseTodosUsuarios))  
29     }  
30
```

MVC – Pesq. usuário por código

- Quando for solicitado ao controlador para que ele retorne uma página contendo o usuário com determinado código, ele:
 - Obtém o código informado pelo usuário (contido no corpo da requisição (dentro de um formulário (tag <form>) contendo o campo “codigo”) (linha 32).
 - Retorna a página resultante da execução da função **views.html.usuario**. Esta função requer dois parâmetros: o código e o objeto da classe Usuario (ou o valor None para indicar que não existe usuário com o código solicitado. (linha 34)

```
31 def pesquiseUsuarioPorCodigo = Action { implicit request =>
32   val codigo = Form("codigo" -> number).bindFromRequest.get
33
34   Ok(views.html.usuario(codigo, crud pesquiseUsuario(codigo)))
35 }
36
```

MVC – Pesquisando ocorrências

- Raciocínio análogo ocorre quando o navegador solicita páginas relacionadas às ocorrências.

```
37 // ocorrencias
38
39 def pesquisarTodasOcorrencias = Action {
40   Ok(views.html.ocorrencias(crud.pesquisarTodasOcorrencias))
41 }
42
43 def pesquisarOcorrenciasPorUsuario = Action { implicit request =>
44   val codigo = Form("codigo" -> number).bindFromRequest.get
45
46   Ok(views.html.ocorrenciasPorUsuario(crud.pesquisarOcorrenciasPorUsuario(codigo)))
47 }
48
```

MVC – Cadastrando Ocorrência

- Ao cadastrar uma ocorrência, o controlador:
 - Extrai os dados da requisição HTTP (linhas 50 e 51). Observar que o campo “quando” é opcional.
 - Instancia uma Ocorrencia com os dados obtidos (linha 52)
 - Solicita que o atributo crud faça o cadastramento e devolva uma resposta (linha 53).
 - Retorna uma página HTML resultante da execução da função views.html.index (linha 55).

```
49 def cadastreOcorrencia = Action { implicit request =>
50   val form = Form(tuple("usuario_codigo" -> number, "quando" -> optional(date), "descricao" -> text))
51   val (usuario_codigo, quando, descricao) = form.bindFromRequest.get
52   val ocorrencia = Ocorrencia(usuario_codigo=usuario_codigo, quando=quando, descricao=descricao)
53   val resposta = crud cadastreOcorrencia(ocorrencia)
54
55   Ok(views.html.index(Some(resposta)))
56 }
```

MVC – Usuario

- Um usuário é caracterizado pelo seu código e pelo seu nome.
- A linha 1 define que a classe faz parte do pacote models.dados.
- A linha 3:
 - Define a classe Usuario
 - Define os atributos codigo e nome
 - Define o método construtor com dois parâmetros
 - Define os métodos “getCodigo” e “getNome”
 - Define que objetos da classe Usuario são imutáveis (como as classes String e Integer em Java).

```
Usuario.scala
1 package models.dados
2
3 case class Usuario(codigo: Int, nome: String)
```

MVC – Ocorrência

- Uma ocorrência é caracterizada pelo seu identificador (atributo id), código do usuário, quando a ocorrência aconteceu e sua descrição.
- O atributo id tem valor default “0”.
- O atributo quando é opcional. Seu valor será ou None ou Some(objeto da classe Date). Em Java seria ou null ou objeto da classe Date.

```
Ocorrencia.scala x
1 package models.dados
2
3 import java.util.Date
4
5 case class Ocorrencia(id: Int = 0, usuario_codigo: Int, quando: Option[Date], descricao: String)
```

MVC – CRUD

- CRUD é o objeto que interaje com o banco de dados.
- Neste exemplo, a interação se dá via biblioteca Anorm (nativa do Play).
- Poderia ter sido usada outra biblioteca Scala (como Slick) ou Java (como Hibernate).

```
CRUD.scala
1 package models
2
3 import models.dados.{Usuario, Ocorrencia}
4
5 import anorm.{SQL, ~}
6 import anorm.SqlParser.{int, str, get}
7
8 import play.api.db.DB
9 import play.api.Play.current
10
11 import java.util.Date
12
13 class CRUD {
14
15     val nomeBD = "demoBD"
16     val parserUsuario = int("codigo") ~ str("nome")
17     val parserOcorrencia = int("id") ~ int("usuario_codigo") ~ get[Option[Date]]("quando") ~ str("descricao")
18
19     def cadastreUsuario(usuario: Usuario) = {
20
21     }
22
23     def pesquiseUsuario(codigo: Int) = {
24
25     }
26
27     def pesquiseTodosUsuarios = {
28
29     }
30
31     def pesquiseTodasOcorrencias = {
32
33     }
34
35     def cadastreOcorrencia(ocorrencia: Ocorrencia) = {
36
37     }
38
39     def pesquiseOcorrenciasPorUsuario(codigo: Int): Either[String, (Usuario, List[Ocorrencia])] = {
40
41     }
42
43     private def retorneOcorrencias(codigo: Int) = {
44
45     }
46
47 }
```


MVC – CRUD

- A classe CRUD se utiliza de:
 - Classes do domínio do problema (linha 3)
 - Objetos e funções da biblioteca Anorm (linhas 5 e 6)
 - Objetos e funções do próprio Play (linhas 8 e 9)
 - Classe Java (linha 11).

```
CRUD.scala x
1 package models
2
3 import models.dados.{Usuario, Ocorrencia}
4
5 import anorm.{SQL, ~}
6 import anorm.SqlParser.{int, str, get}
7
8 import play.api.db.DB
9 import play.api.Play.current
10
11 import java.util.Date
12
13 class CRUD {
14
```

MVC – Definindo Base de Dados

- O atributo nomeBD define qual base de dados é usada por objetos da classe CRUD.

```
13 class CRUD {  
14  
15     val nomeBD = "demoBD"
```

- O valor “demoBD” foi definido no arquivo conf/application.conf:

```
42 db.demoBD.driver=com.mysql.jdbc.Driver  
43 db.demoBD.url="jdbc:mysql://localhost/demoBDPlay"  
44 db.demoBD.user="demoBDUsuario"  
45 db.demoBD.password="demoBDSenha"
```

MVC – Anorm: ResultSet

- Segundo a API JDBC o resultado de uma consulta (select) é armazenado em um objeto da classe ResultSet.
- É relativamente trabalhoso extrair os dados deste objeto. **Anorm simplifica.**
- Linha 16 : define uma função para extrair o código e o nome de uma linha do ResultSet (para Usuario).
- Linha 17: mesma ideia para Ocorrencia. A coluna **quando** pode conter null (neste caso o null é representado por None).

```
16  val parserUsuario = int("codigo") ~ str("nome")
17  val parserOcorrencia = int("id") ~ int("usuario_codigo") ~ get[Option[Date]]("quando") ~ str("descricao")
18
```

MVC – Anorm: Cadastrando Usuário

- Linha 20: pesquisar se já existe usuário cadastrado com código igual ao código do usuário que se deseja cadastrar.

```
19  def cadastreUsuario(usuario: Usuario) = {
20      val usuarioJaCadastrado = pesquiseUsuario(usuario.codigo)
21
22      usuarioJaCadastrado match {
23          case Some(Usuario(_, nome)) => "Já existe usuário com o código " + usuario.codigo + " : " + nome
24          case None => {
25              DB.withConnection(nomeBD) {implicit conn =>
26                  val query =
27                      """
28                      insert into usuarios (codigo, nome) values ({codigo}, {nome})
29                      """
30                  SQL(query).on("codigo" -> usuario.codigo, "nome" -> usuario.nome).executeInsert()
31                  "Usuário cadastrado com sucesso"
32              }
33          }
34      }
35  }
36
```

MVC – Anorm: Cadastrando Usuário

- Linhas 22 e 23: caso já exista usuário, retornar a mensagem informando que já existe. Obs: o método `cadastreUsuario` retorna `String`.

```
19  def cadastreUsuario(usuario: Usuario) = {
20      val usuarioJaCadastrado = pesquiseUsuario(usuario.codigo)
21
22      usuarioJaCadastrado match {
23          case Some(Usuario(_, nome)) => "Já existe usuário com o código " + usuario.codigo + " : " + nome
24          case None => {
25              DB.withConnection(nomeBD) {implicit conn =>
26                  val query =
27                      """
28                      insert into usuarios (codigo, nome) values ({codigo}, {nome})
29                      """
30                  SQL(query).on("codigo" -> usuario.codigo, "nome" -> usuario.nome).executeInsert()
31                  "Usuário cadastrado com sucesso"
32              }
33          }
34      }
35  }
36
```

MVC – Anorm: Cadastrando Usuário

- Linhas 26 a 29: comando SQL para inserir dados na tabela usuarios.
- Linha 30: método “on” substitui os parâmetros {codigo} e {nome} pelos seus respectivos valores e, em seguida, executa-se a consulta.
- Linha 31: retorna a mensagem.

```
24 case None => {  
25     DB.withConnection(nomeBD) {implicit conn =>  
26         val query =  
27             """  
28             insert into usuarios (codigo, nome) values ({codigo}, {nome})  
29             """  
30         SQL(query).on("codigo" -> usuario.codigo, "nome" -> usuario.nome).executeInsert()  
31         "Usuário cadastrado com sucesso"  
32     }  
33 }
```

MVC – Anorm: Pesquisando Usuário

- Linha 44: após substituir o parâmetro {codigo} pelo valor da variável codigo, a query é executada e o resultado (ResultSet) é analisado pelo parserUsuario. Este retornará None (caso não exista usuário com o código solicitado) ou Some(codigo~nome).
- Linha 45: retorna ou Some(Usuario) ou None dependendo do

```
37     def pesquisaUsuario(codigo: Int) = {
38
39         DB.withConnection(nomeBD) {implicit conn =>
40             val query =
41                 """
42                 select codigo,nome from usuarios where codigo = {codigo}
43                 """
44             val result = SQL(query).on("codigo" -> codigo).as(parserUsuario.singleOpt)
45
46             result map {case codigo~nome => Usuario(codigo, nome)}
47         }
48     }
49
```

MVC – Anorm: Pesquisando Todos os Usuários

- Linha 57: executa a consulta SQL (linha 55) e converte o ResultSet em uma lista contendo código e nome.
- Linha 59 : retorna uma lista contendo objetos da classe Usuario.

```
50     def pesquiseTodosUsuarios = {  
51  
52         DB.withConnection(nomeBD) {implicit conn =>  
53             val query =  
54                 """  
55                 select codigo,nome from usuarios order by nome  
56                 """  
57             val result = SQL(query).as(parserUsuario *)  
58  
59             result map {case codigo~nome => Usuario(codigo, nome)}  
60         }  
61     }  
62
```


MVC – Anorm: Pesquisando Todas as Ocorrências

- A mesma ideia para retornar uma lista com objetos da classe Ocorrencia.

```
63     def pesquiseTodasOcorrencias = {  
64  
65         DB.withConnection(nomeBD) {implicit c =>  
66             val query =  
67                 """  
68                 select id,usuario_codigo,quando,descricao  
69                     from ocorrencias order by quando  
70                 """  
71  
72             val result = SQL(query).as(parserOcorrencia *)  
73  
74             result map {case id~usuario_codigo~quando~descricao =>  
75                 Ocorrencia(id,usuario_codigo,quando,descricao)}  
76         }  
77     }  
78
```

MVC – Anorm: Cadastrando uma Ocorrência

- O método retorna uma String como resposta.

```
79 def cadastreOcorrencia(ocorrencia: Ocorrencia) = {
80     val usuario = pesquiseUsuario(ocorrencia.usuario_codigo)
81
82     usuario match {
83         case None => "Não existe usuário com o código " + ocorrencia.usuario_codigo
84         case Some(Usuario(codigo,_)) => {
85             DB.withConnection(nomeBD) { implicit conn =>
86                 val query =
87                     """
88                     insert into ocorrencias (usuario_codigo, quando, descricao)
89                     values ({usuario_codigo}, {quando}, {descricao})
90                     """
91                 SQL(query).on("usuario_codigo" -> ocorrencia.usuario_codigo,
92                     "quando" -> ocorrencia.quando, "descricao" -> ocorrencia.descricao).executeInsert()
93                 "Ocorrencia cadastrada com sucesso!"
94             }
95         }
96     }
97 }
98
```

MVC – Anorm: Pesquisando Ocorrências de um Usuário

- O método retorna:
 - Ou uma String indicando que não existe usuário com o código informado (linha 103).
 - Ou uma tupla (Usuario, List[Ocorrencia]) contendo um usuário e uma lista de ocorrências (linha 106).
 - Scala: Either é semelhante a Option. A diferença é que retorna ou Left ou Right.

```
99  def pesquiseOcorrenciasPorUsuario(codigo: Int): Either[String, (Usuario, List[Ocorrencia])] = {
100    val usuario = pesquiseUsuario(codigo)
101
102    usuario match {
103      case None => Left("Não existe usuário com o código " + codigo)
104      case Some(usuario) => {
105        val ocorrencias = retorneOcorrencias(codigo)
106        Right((usuario, ocorrencias))
107      }
108    }
109  }
110
```

MVC – Anorm: Pesquisando Ocorrências de um Usuário

- O método auxiliar (privado) para retornar uma lista de ocorrências (dado do tipo List[Ocorrencia]).

```
111     private def retorneOcorrencias(codigo: Int) = {
112         DB.withConnection(nomeBD) { implicit conn =>
113             val query =
114                 """
115                 select * from ocorrencias where usuario_codigo = {codigo} order by quando
116                 """
117             val result = SQL(query).on("codigo" -> codigo).as(parserOcorrencia *)
118             result map {case id~usuario_codigo~quando~descricao =>
119                 Ocorrencia(id,usuario_codigo,quando,descricao)}
120         }
121     }
```

MVC – Página Inicial

- O arquivo **app/views/index.scala.html** define a página inicial:
 - Possui um parâmetro opcional cujo valor default é None (linha 1).
 - Monta o cabeçalho padrão presente em todas as páginas (linha 3).
 - Se a mensagem estiver definida mostra o seu conteúdo (linhas 5 e 6).

```
index.scala.html x
1 @(msg: Option[String] = None)
2
3 @tags.cabecalho("Usuários e Ocorrências")
4
5 @if(msg.isDefined) {
6     <h3>Resultado: @msg</h3>
7 }
8
```

MVC – Página Inicial (cont.)

- Pesquisa e cadastro de usuários.
- Linhas 11, 12 e 18: gera os links de acordo com conf/routes

```
9  <h3>Usuários</h3>
10 <ul>
11   <li><a href="@routes.Application.pesquiseTodosUsuarios">Mostrar Cadastrados</a></li>
12   <li><form action="@routes.Application.pesquiseUsuarioPorCodigo" method="get">
13       Mostrar com código:
14       <input type="text" name="codigo" size=5><br>
15       <input type="submit" value="Mostrar">
16   </form>
17 </li>
18   <li><form action="@routes.Application.cadastreUsuario" method="post">
19       Cadastrar novo:<br>
20       Código: <input type="text" name="codigo" size="5"><br>
21       Nome: <input type="text" name="nome" size=50><br>
22       <input type="submit" value="Cadastrar">
23   </form>
24 </li>
25 </ul>
26
```

MVC – Página Inicial (cont.)

- Mesma ideia para cadastro e pesquisa de ocorrências.

```
27 <h3>Ocorrências</h3>
28
29 <ul>
30   <li><a href="@routes.Application.pesquiseTodasOcorrencias">Mostrar Cadastradas</a></li>
31   <li><form action="@routes.Application.pesquiseOcorrenciasPorUsuario" method="get">
32     Mostrar associadas ao Usuário código:
33     <input type="text" name="codigo" size=5><br>
34     <input type="submit" value="Mostrar">
35   </form>
36 </li>
37   <li><form action="@routes.Application.cadastreOcorrencia" method="post">
38     Cadastrar nova:<br>
39     Código do Usuário: <input type="text" name="usuario_codigo" size="5"> <br>
40     Data : <input type="text" name="quando" size="8"><br>
41     Descrição: <input type="text" name="descricao" size="40"><br>
42     <input type="submit" value="Cadastrar">
43   </form>
44 </li>
45 </ul>
```


MVC – Página Usuário

- O arquivo **app/views/usuario.scala.html** mostra os dados de um usuário ou uma mensagem informando que não existe usuário com aquele código.

```
usuario.scala.html x
1 @(codigo: Int, usuario: Option[models.dados.Usuario])
2
3 @tags.cabecalho("Usuário por Código")
4
5 @usuario match {
6     case None => {@semUsuario(codigo)}
7     case Some(u) => {@mostreUsuario(u)}
8 }
9
10 @semUsuario(codigo: Int) = {
11     <h2>Não há usuário</h2>
12     <p>Nenhum usuário cadastrado com código @codigo</p>
13 }
14
15 @mostreUsuario(usuario: models.dados.Usuario) = {
16     <p>Nome: @usuario.nome Código: @usuario.codigo</p>
17 }
```


MVC – Página Usuários

- O arquivo **app/views/usuarios.scala.html** mostra os dados de todos os usuário ou uma mensagem informando que não há usuários cadastrados.

```
usuarios.scala.html x
1 @(usuarios: List[models.dados.Usuario])
2
3 @tags.cabecalho("Usuários Cadastrados")
4
5 @usuarios.size match {
6   case 0 => {<h3>Não há usuários cadastrados</h3>}
7   case _ => {
8     <table border="1">
9       <tr><th>Código</th><th>Nome</th></tr>
10      @for( usuario <- usuarios) {
11        <tr><td>@usuario.codigo</td> <td>@usuario.nome</td></tr>
12      }
13    }
14  }
```

MVC – Página Ocorrências

- O arquivo **app/views/ocorrencias.scala.html** mostra os dados de todas as ocorrências.

```
ocorrencias.scala.html x
1 @(ocorrencias: List[models.dados.Ocorrencia])
2
3 @tags.cabecalho("Ocorrências Cadastradas")
4
5 @tags.mostraOcorrencias(ocorrencias)
```

MVC – Página Ocorrências por Usuário

- O arquivo `app/views/ocorrenciasPorUsuario.scala.html` mostra os dados de todas as ocorrências.
- Observe o uso da classe `Either` (Scala).

```
ocorrenciasPorUsuario.scala.html x
1  @(resultado: Either[String, (models.dados.Usuario, List[models.dados.Ocorrencia])])
2
3  @tags.cabecalho("Ocorrências de Usuário")
4
5  @resultado match {
6      case Left(msg) => {<h3 style="color: red">@msg</h3>}
7      case Right((usuario, ocorrencias)) => {
8          <h3> Usuário: @usuario.nome </h3>
9
10         @tags.mostraOcorrencias(ocorrencias)
11     }
12 }
```

MVC – Tag cabeçalho

- O arquivo `app/views/tags/cabecalho.scala.html` gera um fragmento de código HTML e não uma página inteira.
- Seu objetivo é montar um cabeçalho para ser usado pelas páginas da aplicação.

```
cabecalho.scala.html *  
1 @(titulo : String)  
2  
3 <h1>INE5646 - demoBD</h1>  
4 <h2>@titulo</h2>
```

MVC – Tag mostraOcorrencias

- O arquivo `app/views/tags/mostraOcorrencias.scala.html` também gera um fragmento de código HTML.
- Seu objetivo é mostrar, em uma tabela, um conjunto de ocorrências.

```
mostraOcorrencias.scala.html x
1  @(ocorrencias: List[models.dados.Ocorrencia])
2
3  @ocorrencias.size match {
4      case 0 => {<p>Nenhuma ocorrência registrada!</p>}
5      case n => {
6          <table border="1">
7              <tr><th>Id</th><th>Cód. Usuário</th><th>Data</th><th>Descrição</th>
8              @for( ocorrencia <- ocorrencias) {
9                  <tr>
10                     <td>@ocorrencia.id</td>
11                     <td>@ocorrencia.usuario_codigo</td>
12                     <td>@ocorrencia.quando</td>
13                     <td>@ocorrencia.descricao</td>
14                 </tr>
15             }
16         }
17     }
```

Serviços Web

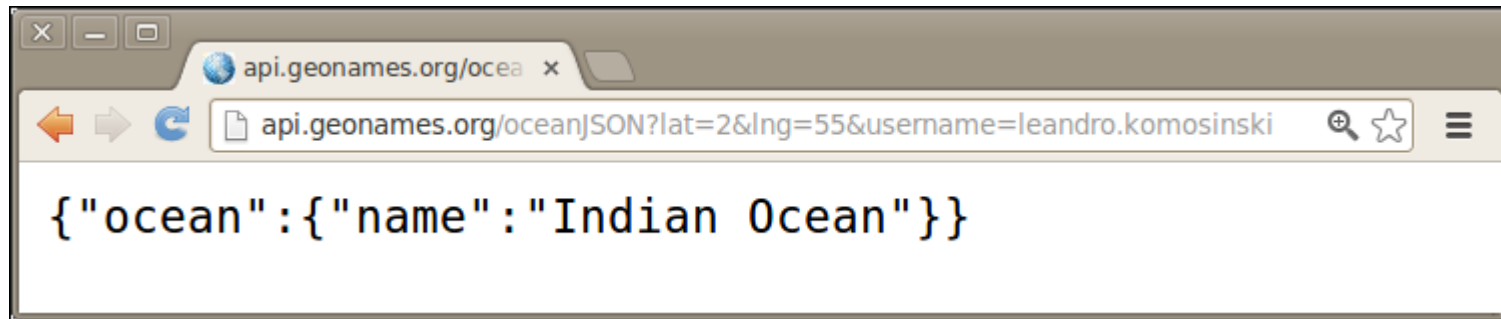
- O Play possui um **objeto** chamado **WS** definido no pacote **play.api.libs.ws** dedicado a processamento de serviços web.
- Este objeto funciona como um cliente HTTP para realizar a comunicação com o provedor do serviço.
- Coerentemente, o processamento de um serviço (acesso ao provedor do serviço) é realizado assíncronamente.
- Não há suporte nativo para SOAP (deve-se usar alguma biblioteca Java).
- O suporte para REST é muito bom (embora a documentação oficial seja, no momento, escassa).

Serviços Web - Exemplo

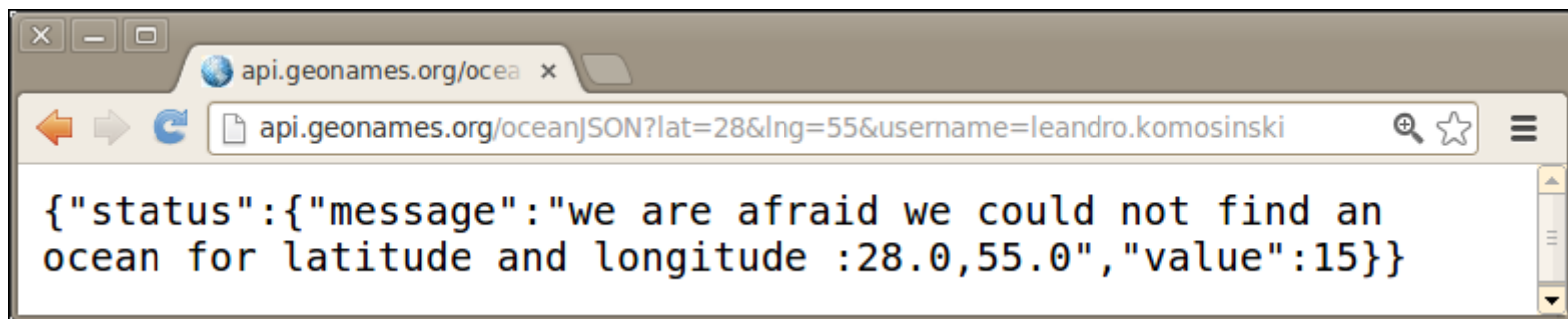
- A apresentação das facilidades para processamento de serviços web é feita por meio de um exemplo:
 - O site <http://api.geonames.org> disponibiliza diversos serviços relacionados à geolocalização.
 - Um dos serviços, usado no exemplo, é o que permite identificar, a partir de uma coordenada (latitude e longitude), se existe ou não algum oceano. Se existir, o serviço retorna o nome do oceano.

Serviços Web - Exemplo

- Se a coordenada estiver em algum oceano, o serviço retorna um objeto JSON como mostra a figura abaixo.



- Se não contiver, retorna um objeto JSON como mostra a figura abaixo.



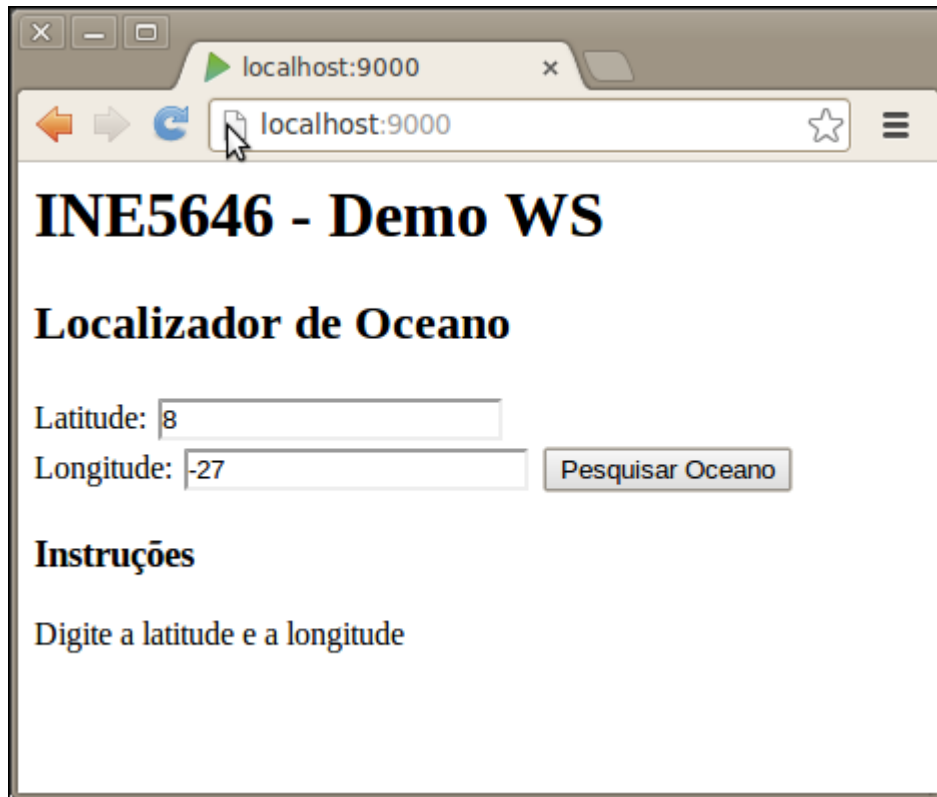
Exemplo – Página Inicial

- Página inicial da aplicação exemplo

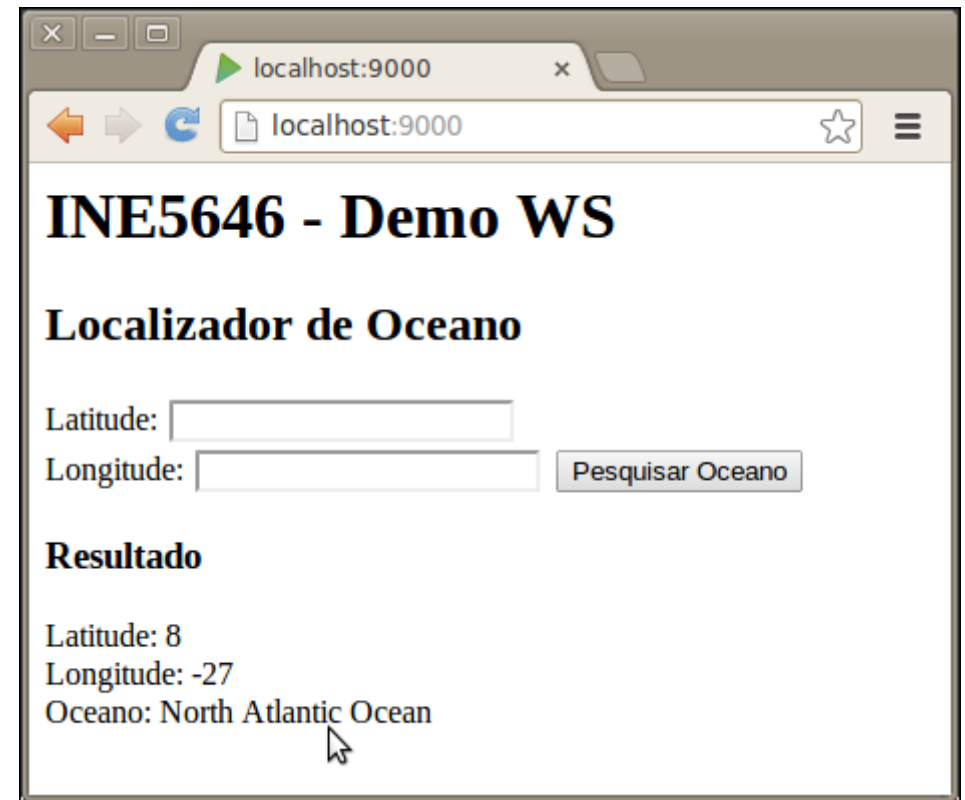


Pesquisando por oceano existente

- Pesquisando na latitude 8 e na longitude -27



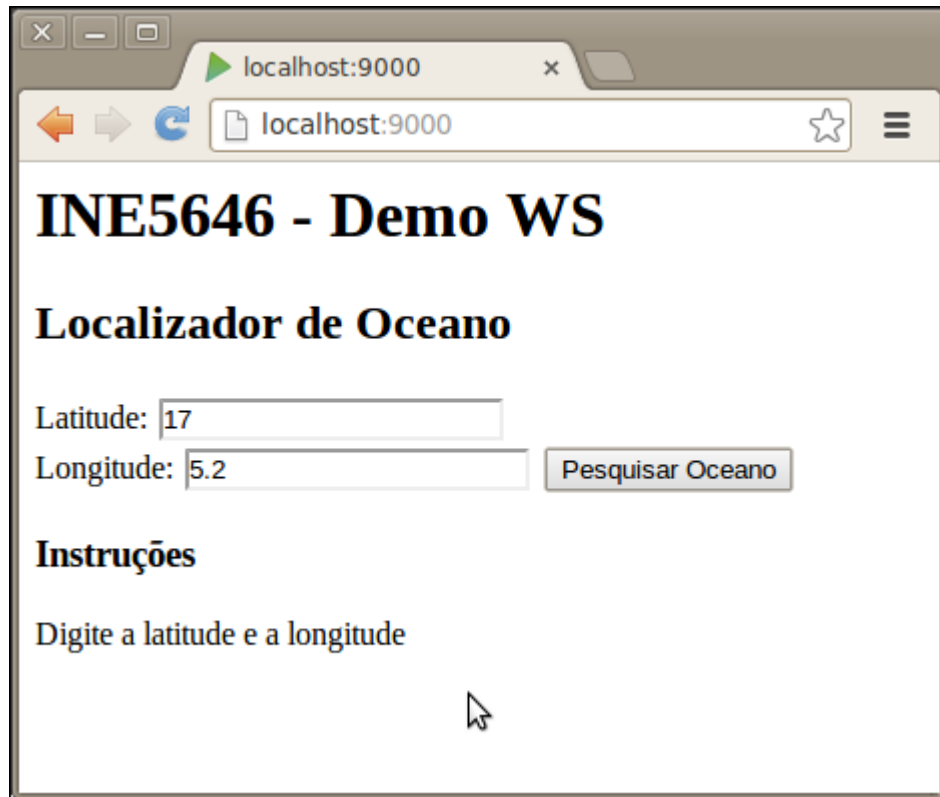
A screenshot of a web browser window showing a web application titled "INE5646 - Demo WS". The page has a section titled "Localizador de Oceano" with two input fields: "Latitude:" containing the value "8" and "Longitude:" containing the value "-27". To the right of these fields is a button labeled "Pesquisar Oceano". Below this section is another section titled "Instruções" with the text "Digite a latitude e a longitude". The browser's address bar shows "localhost:9000".



A screenshot of the same web browser window after the search has been executed. The "Localizador de Oceano" section now has empty input fields for "Latitude:" and "Longitude:". The "Pesquisar Oceano" button is still present. Below this, a new section titled "Resultado" displays the search results: "Latitude: 8", "Longitude: -27", and "Oceano: North Atlantic Ocean". The browser's address bar remains "localhost:9000".

Pesquisando por oceano inexistente

- Pesquisando na latitude 17 e na longitude 5.2



localhost:9000

INE5646 - Demo WS

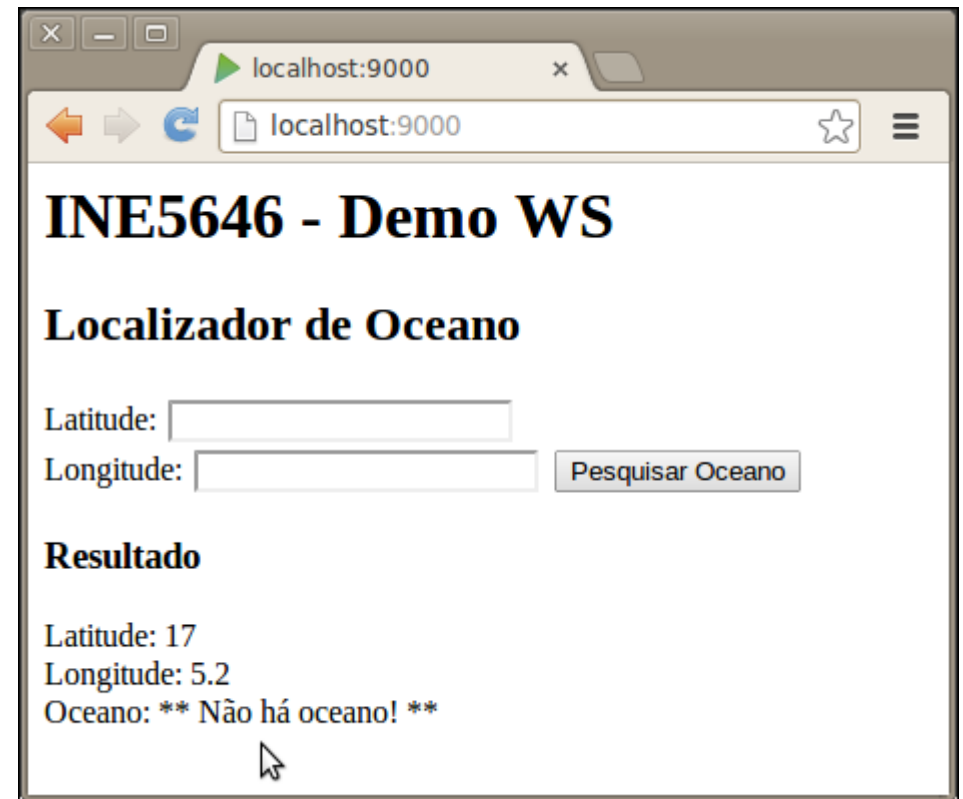
Localizador de Oceano

Latitude:

Longitude:

Instruções

Digite a latitude e a longitude



localhost:9000

INE5646 - Demo WS

Localizador de Oceano

Latitude:

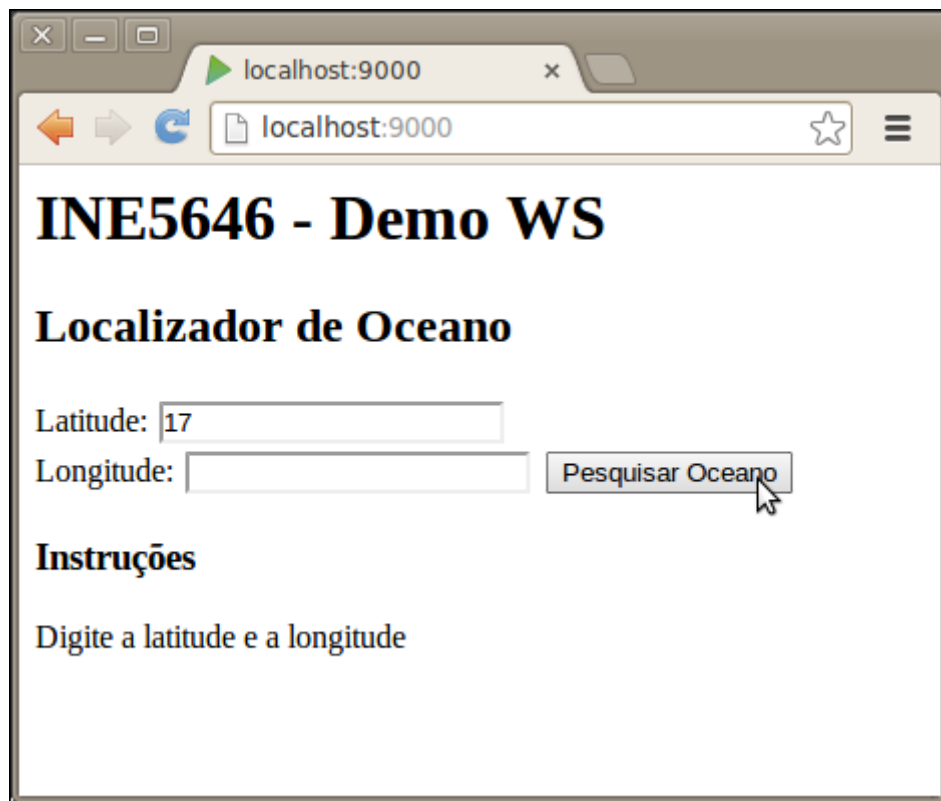
Longitude:

Resultado

Latitude: 17
Longitude: 5.2
Oceano: ** Não há oceano! **

Pesquisando sem longitude

- Pesquisando na latitude 17 e sem longitude.



localhost:9000

INE5646 - Demo WS


Localizador de Oceano

Latitude:

Longitude:

Instruções

Digite a latitude e a longitude



localhost:9000

INE5646 - Demo WS

Localizador de Oceano

Latitude:

Longitude:

Instruções

Digite os dois campos

MVC – página index.scala.html

- Observe o uso do Either (Scala)

```
index.scala.html x
1  @(resposta: Either[String, (String,String,String)])
2  <h1> INE5646 - Demo WS</h1>
3  <h2>Localizador de Oceano</h2>
4  <form action="@routes.Application.pesquisa" method="post">
5      Latitude: <input type="text" name="lat"><br>
6      Longitude: <input type="text" name="long">
7      <input type="submit" value="Pesquisar Oceano">
8  </form>
9
10 @resposta match {
11     case Left(msg) => {
12         <h3>Instruções</h3>
13         @msg
14     }
15     case Right((lat, long, oceano)) => {
16         <h3>Resultado</h3>
17         <p>Latitude: @lat <br>
18             Longitude: @long <br>
19             Oceano: @oceano</p>
20     }
21 }
```

MVC – conf/routes

- Observe que a mesma URL (“/”) terá tratamento diferente em função do tipo da requisição (GET ou POST).

```
routes x
1 GET / controllers.Application.index
2
3 POST / controllers.Application.pesquisa
```

MVC – Objeto Application

```
Application.scala  x
1  package controllers
2
3  import play.api.data.Form
4  import play.api.data.Forms.{text, tuple, optional}
5  import play.api.mvc.{Controller, Action}
6  import play.api.libs.ws.WS
7  import play.api.libs.concurrent.Execution.Implicits.defaultContext
8
9  object Application extends Controller {
10
11      def index = Action { ...
12
13      }
14
15      def pesquise = Action { implicit request => ...
16
17      }
18
19      private def processe(lat: String, long: String) = { ...
20
21      }
22
23      private def definaEndereco(lat: String, long: String) = { ...
24
25      }
26
27      }
28
29  }
```

MVC – Objeto Application

- O papel do controlador, neste exemplo, é:
 - Receber a requisição
 - Executar assincronamente o web service.
 - Montar a página a ser devolvida ao browser.

MVC – Application.index

- Ao executar o método index, retorna a página gerada pela função **views.html.index** passando como parâmetro a mensagem “Digite a latitude e a longitude”.

```
Application.scala x
1 package controllers
2
3 import play.api.data.Form
4 import play.api.data.Forms.{text, tuple, optional}
5 import play.api.mvc.{Controller, Action}
6 import play.api.libs.ws.WS
7 import play.api.libs.concurrent.Execution.Implicits.defaultContext
8
9 object Application extends Controller {
10
11     def index = Action {
12         Ok(views.html.index(Left("Digite a latitude e a longitude")))
13     }
14 }
```

MVC – Application.pesquisa

- Ao executar o método pesquisa:
 - Obtém os dados digitados pelo usuário (linhas 16 e 17).
 - Caso o usuário tenha informado **a latitude E a longitude** então processa estes dados (linha 19).
 - Caso contrário, retorna a página inicial com a mensagem “Digite os dois campos” (linha 20).

```
15  def pesquisa = Action { implicit request =>
16      val form = Form(tuple("lat" -> optional(text), "long" -> optional(text)))
17      val (optLat,optLong) = form.bindFromRequest.get
18      (optLat, optLong) match {
19          case (Some(lat), Some(long)) => processe(lat,long)
20          case _ => Ok(views.html.index(Left("Digite os dois campos")))
21      }
22  }
23
```

MVC – Application.processe

- Ao executar o método processe:
 - Instancia o serviço (linha 25).
 - Assincronamente, invoca o serviço (linha 27) e imediatamente obtém futureResposta. Este objeto, em algum momento futuro, conterá a resposta enviada pelo provedor do serviço.

```
24 private def processe(lat: String, long: String) = {
25     val servico = WS.url(definaEndereco(lat, long))
26     Async {
27         val futureResposta = servico get // método HTTP GET
28
29         futureResposta map { resposta =>
30             val jsonValue = (resposta.json \ "ocean" \ "name")
31             val oceano = jsonValue.asOpt[String].getOrElse("** Não há oceano! **")
32             Ok(views.html.index(Right(lat, long, oceano)))
33         }
34     }
35 }
36
```

MVC – Application.processe (cont)

- Quando a resposta chegar (linha 29):
 - Transforme a resposta para o formato JSON e obtenha o valor do atributo “ocean” e, deste, obtenha o valor do atributo “name” (linha 30):
 - Quando o serviço retorna um oceano este é descrito como, por exemplo, {“ocean”: {“name”: “Indian Ocean”}}
 - Obtenha o nome do oceano (String) contido na variável jsonValue ou então a mensagem “** Não há oceano! **” (linha 31).

```
24 private def processe(lat: String, long: String) = {
25     val servico = WS.url(definaEndereco(lat, long))
26     Async {
27         val futureResposta = servico.get // método HTTP GET
28
29         futureResposta.map { resposta =>
30             val jsonValue = (resposta.json \ "ocean" \ "name")
31             val oceano = jsonValue.asOpt[String].getOrElse("** Não há oceano! **")
32             Ok(views.html.index(Right(lat, long, oceano)))
33         }
34     }
35 }
36
```

MVC – Application.processe (cont)

- Retorne ao browser a página gerada pela função `views.html.index` passando como parâmetro a latitude, a longitude e o nome do oceano (ou a mensagem de que não há oceano) (linha 32).
- **FUNDAMENTAL**: o método `processe` é executado em poucos milisegundos, mesmo que o serviço demore alguns segundos para retornar a resposta! (linha 26)

```
24 private def processe(lat: String, long: String) = {
25     val servico = WS.url(definaEndereco(lat, long))
26     Async {
27         val futureResposta = servico.get // método HTTP GET
28
29         futureResposta.map { resposta =>
30             val jsonValue = (resposta.json \ "ocean" \ "name")
31             val oceano = jsonValue.asOpt[String].getOrElse("** Não há oceano! **")
32             Ok(views.html.index(Right(lat, long, oceano)))
33         }
34     }
35 }
36
```

MVC – Application.definaEndereco

- Como o serviço é do tipo RESP, basta definir qual a URL da consulta (linha 39):
 - Host: api.geonames.org
 - Nome do serviço: oceanJSON
 - Parâmetros: latitude, longitude e nome do usuário (previamente cadastrado para liberar o acesso gratuito ao serviço).

```
37 private def definaEndereco(lat: String, long: String) = {  
38     val u = "leandro.komosinski"  
39     s"http://api.geonames.org/oceanJSON?lat=$lat&lng=$long&username=$u"  
40 }
```

- Linha 39: (Scala) interpolação de String (introduzido na versão 2.10.0): substitui, na string, as expressões precedidas por “\$” pelos valores das variáveis. No caso, \$lat, \$long e \$u.

Stream de dados

- O desempenho de uma aplicação para web, como em qualquer aplicação distribuída, é afetado:
 - Pelo volume de dados que trafega entre as camadas 1 (browser) e 2 (servidor).
 - Tempo de processamento na camada 2.
 - Volume de recursos consumidos (tipicamente memória) na camada 2.
 - Quantidade de comunicações (requisições) entre as camadas 1 e 2
- **No contexto web 1.0**, o predomínio é de aplicações com:
 - Baixo volume de dados. Envia-se poucos dados contidos em formulário e retorna-se o conteúdo de páginas.
 - Requisições processadas rapidamente, sem depender de sistemas externos.
 - Sessão (associada a cada usuário simultâneo) ocupando pouca memória.
 - Ritmo das comunicações associado à velocidade de digitação do usuário.

Stream de dados

- **No contexto web 2.0**, o predomínio é de aplicações com:
 - Alto volume de dados (em alguns casos). Usuário envia arquivo; servidor envia arquivo ou dados em tempo real (live streaming).
 - Requisições processadas demoradamente (processamento de todo o volume de dados e acesso a sistemas externos (web services)).
 - A sessão (associada a cada usuário simultâneo) ocupando muita memória.
 - Alto ritmo de comunicações (via Ajax).

Stream de dados

- A expressão “stream de dados” refere-se ao contexto onde o volume de dados transmitido entre as camadas 1 e 2, nos dois sentidos, é alto.
- Tipicamente, a expressão aparece em tarefas de upload/download de arquivo e de transmissão ao vivo de dados (*live streaming*).
- O conceito de stream de dados, ainda que útil/necessário, tem alto potencial para limitar a escalabilidade de uma aplicação para web.

Stream de dados - upload

- Cenário típico:
 - Cada usuário fazendo upload de arquivo para ser processado no servidor (camada 2).
 - Cada arquivo é armazenado na memória e só então processado.
 - Solução claramente não escalável.
- Abordagem Play:
 - Utiliza um *iteratee* para consumir os dados do arquivo e ir armazenando-os em um arquivo temporário.
 - Iteratee: versão funcional do “iterator de Java”
 - Sobre iteratees:
<http://mandubian.com/2012/08/27/understanding-play2-iteratees-for-normal-humans/>

Stream de dados - download

- Há duas situações típicas:
 - (1) Download de dados com tamanho conhecido (ex: download de arquivo)
 - (2) Download de dados sem tamanho conhecido (ex: transmissões ao vivo – *live streaming*)
- Cenário típico (1):
 - Para cada usuário, o servidor (camada 2) precisa enviar ao cliente (camada 1) o conteúdo de um arquivo como resposta à requisição HTTP.
 - Cada arquivo é armazenado na memória e só então enviado.
 - Solução claramente não escalável.
- Cenário típico (2):
 - A resposta a ser enviada para cada usuário, por definição, nunca está completa.
 - A resposta deve ser dividida em partes (chunks). Enviar cada parte a medida que está disponível.

Stream de dados - download

- O protocolo HTTP 1.1 já previu como tratar os dois cenários:
 - Cenário (1): na resposta, o servidor deve incluir o tamanho do arquivo, definindo um valor para o cabeçalho **Content-Length**.
 - Cenário (2): na resposta, o servidor deve incluir o cabeçalho **Transfer-Encoding**. Com isso indica que a resposta será enviada em partes (*chunks*). Para cada parte enviada indica-se o tamanho, em bytes, da parte.
 - http://en.wikipedia.org/wiki/Chunked_transfer_encoding
- Abordagem Play:
 - Utiliza um *Enumerator* para gerar *chunks* de conteúdo.
 - Enumerator: produtor de dados.
 - Iteratee: consumidor de dados produzidos por algum Enumerator.

Stream de dados - Exemplo

- No exemplo, o usuário pode:
 - Fazer upload do arquivo. O servidor responde informando o nome do arquivo, o tipo (mime-type) do arquivo e onde o arquivo foi salvo.
 - Fazer download de um arquivo (stream de tamanho fixo)
 - Fazer o download de uma página de um site sem conhecer previamente seu tamanho (simulando live streaming).

Exemplo: página inicial



Exemplo: enviando arquivo

- Selecionado o arquivo projeto_musicas.zip



Exemplo: enviando arquivo

- Observar que o arquivo projeto_musicas.zip foi armazenado, no servidor, no diretório /tmp com o nome multipartBody66...09asTemporaryFile.



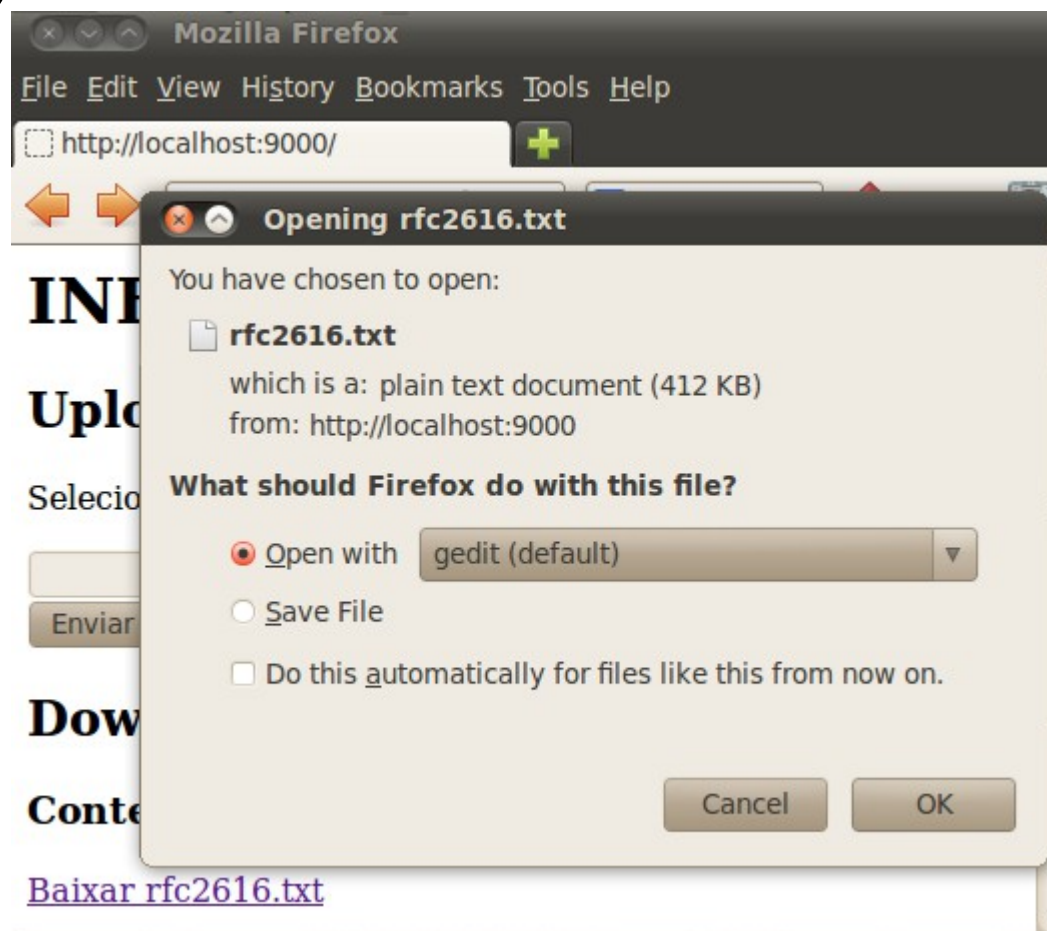
Exemplo: download de arquivo

- Observar a URL solicitada:
localhost:9000/downloadFixo



Exemplo: download de arquivo

- Como resultado, o usuário é informado do arquivo recebido (rfc2616.txt) e seu tamanho (412 KB).



Exemplo: download de stream

- Observar a URL solicitada:
localhost:9000/downloadLive



Exemplo: download de stream

- Como resultado, a página do INE é exibida.



Fontes - MVC: index.scala.html

- Template (1/2) para a página inicial da aplicação.

```
index.scala.html x
1  @(dadosUpload: Option[(String,String,String)] = None)
2
3  <h1>INE5646 - Demo Stream</h1>
4
5  <h2>Upload</h2>
6  @dadosUpload match {
7      case None => {
8          <p>Selecione e envie um arquivo.</p>
9      }
10     case Some((nomeArqEnviado, tipoArq, nomeArqSalvo)) => {
11         <h3>Dados do arquivo enviado</h3>
12         <p>Nome arquivo enviado: @nomeArqEnviado<br>
13             Tipo: @tipoArq<br>
14             Nome arquivo salvo : @nomeArqSalvo
15         </p>
16     }
17 }
18
19 @helper.form(action=routes.Application.upload,
20             'enctype -> "multipart/form-data") {
21     <input type="file" name="arquivo" value="Arquivo..." size="40">
22     <input type="submit" value="Enviar">
23 }
24
```

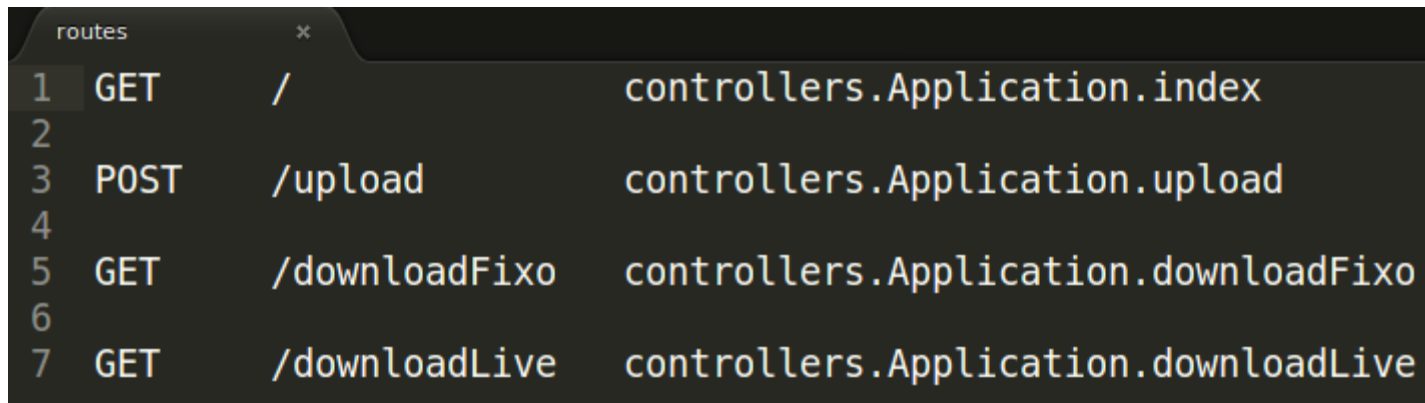
Fontes - MVC: index.scala.html

- Template (2/2) para a página inicial da aplicação.

```
25 <h2>Download</h2>
26
27 <h3>Conteúdo fixo</h3>
28 <a href="@routes.Application.downloadFixo">Baixar rfc2616.txt</a>
29
30 <h3>Conteúdo variável (streaming)</h3>
31 <p>(Simulação)</p>
32 <a href="@routes.Application.downloadLive">Site do INE/CTC/UFSC</a>
```

Fontes - MVC: conf/routes

- O arquivo routes define como cada requisição será tratada.

A screenshot of a code editor window titled 'routes' with a close button 'x'. The editor contains a list of seven routes, each on a new line and numbered 1 through 7 in the left margin. Each route consists of an HTTP method, a path, and a controller action, separated by spaces.

1	GET	/	controllers.Application.index
2			
3	POST	/upload	controllers.Application.upload
4			
5	GET	/downloadFixo	controllers.Application.downloadFixo
6			
7	GET	/downloadLive	controllers.Application.downloadLive

Fontes - MVC: Application.java

- Visão geral do controlador.

```
Application.scala x
1 package controllers
2
3 import play.api.mvc.{Controller,Action}
4 import play.api.libs.iteratee.Enumerator
5
6 object Application extends Controller {
7
8     def index = Action { ...
9
10 }
11
12 def upload = Action(parse.multipartFormData) {request => ...
13
14 }
15
16 def downloadFixo = Action { ...
17
18 }
19
20 def downloadLive = Action { ...
21
22 }
23
24 }
```


Fontes - MVC: Application.java

- O método index do controlador Application retorna uma página HTML gerada pela execução da função views.html.index.

```
8  def index = Action {
9    Ok(views.html.index())
10 }
11
```

```
1  @(dadosUpload: Option[(String,String,String)] = None)
2
3  <h1>INE5646 - Demo Stream</h1>
4
5  <h2>Upload</h2>
6  @dadosUpload match {
7    case None => {
8      <p>Selecione e envie um arquivo.</p>
9    }
10   case Some((nomeArqEnviado, tipoArq, nomeArqSalvo)) => {
11     <h3>Dados do arquivo enviado</h3>
12     <p>Nome arquivo enviado: @nomeArqEnviado<br>
13       Tipo: @tipoArq<br>
14       Nome arquivo salvo : @nomeArqSalvo
15     </p>
16   }
17 }
18
19 @helper.form(action=routes.Application.upload,
20               'enctype -> "multipart/form-data") {
21   <input type="file" name="arquivo" value="Arquivo..." size="40">
22   <input type="submit" value="Enviar">
23 }
24
25 <h2>Download</h2>
26
27 <h3>Conteúdo fixo</h3>
28 <a href="@routes.Application.downloadFixo">Baixar rfc2616.txt</a>
29
30 <h3>Conteúdo variável (streamming)</h3>
31 <p>(Simulação)</p>
32 <a href="@routes.Application.downloadLive">Site do INE/CTC/UFSC</a>
```



Fontes - MVC: Application.java

- O método upload do controlador Application:
 - Sabe que a requisição contém um form multi-part (linha 12).
 - Extrai o arquivo contido na requisição (linha 14).

```
12  def upload = Action(parse.multipartFormData) {request =>
13      val OkOption =
14          request.body.file("arquivo").map { arquivo =>
15              val nomeArqEnviado = arquivo.filename
16              val tipoDoArquivo = arquivo.contentType.get
17              val arquivoSalvo: java.io.File = arquivo.ref.file
18              val pathArqSalvo = arquivoSalvo.getAbsolutePath
19              val resposta = Some(nomeArqEnviado, tipoDoArquivo, pathArqSalvo)
20              Ok(views.html.index(resposta))
21          }
22      OkOption.get
23  }
24
```

Fontes - MVC: Application.java

- O método upload do controlador Application:
 - Obtém o nome e o tipo do arquivo enviado (linhas 15 e 16).
 - Tem acesso ao arquivo enviado. Note a integração com Java (linha 17).
 - A variável arquivoSalvo é um objeto Java. (linha 18).

```
12  def upload = Action(parse.multipartFormData) {request =>
13      val OkOption =
14          request.body.file("arquivo").map { arquivo =>
15              val nomeArqEnviado = arquivo.filename
16              val tipoDoArquivo = arquivo.contentType.get
17              val arquivoSalvo: java.io.File = arquivo.ref.file
18              val pathArqSalvo = arquivoSalvo.getAbsolutePath
19              val resposta = Some(nomeArqEnviado, tipoDoArquivo, pathArqSalvo)
20              Ok(views.html.index(resposta))
21          }
22      OkOption.get
23  }
24
```

Fontes - MVC: Application.java

- O método upload do controlador Application:
 - Gera, como resposta, a página inicial passando como parâmetro (variável resposta) os dados do arquivo enviado (linhas 19 e 20).

```
12  def upload = Action(parse.multipartFormData) {request =>
13      val OkOption =
14          request.body.file("arquivo").map { arquivo =>
15              val nomeArqEnviado = arquivo.filename
16              val tipoDoArquivo = arquivo.contentType.get
17              val arquivoSalvo: java.io.File = arquivo.ref.file
18              val pathArqSalvo = arquivoSalvo.getAbsolutePath
19              val resposta = Some(nomeArqEnviado,tipoDoArquivo,pathArqSalvo)
20              Ok(views.html.index(resposta))
21          }
22      OkOption.get
23  }
24
```

Fontes - MVC: Application.java

- O método `downloadFixo` do controlador `Application`:
 - Envia, como resposta, o arquivo “rfc2616.txt”.
 - O método **`sendFile`** se encarrega de calcular o tamanho do arquivo e incluir o cabeçalho `Content-Length` na resposta.

```
25     def downloadFixo = Action {  
26         val nomeArq = "arquivosDownload/rfc2616.txt"  
27         val arquivo = new java.io.File(nomeArq)  
28  
29         Ok.sendFile(arquivo)  
30     }  
31
```

Fontes - MVC: Application.java

- O método `downloadLive` do controlador `Application`:
 - Envia, como resposta, o conteúdo do site `www.inf.ufsc.br`.
 - O Enumerator `e` gerará os *chunks* de dados a partir do conteúdo da página inicial do site acima (linha 34).
 - O método **stream** inclui na resposta o cabeçalho “Transfer-Encoding” e consome os dados gerados pelo Enumerator “e” (linha 36).

```
32     def downloadLive = Action {  
33         val url = new java.net.URL("http://www.inf.ufsc.br")  
34         val e = Enumerator.fromStream(url.openStream())  
35  
36         Ok.stream(e).as("text/html")  
37     }
```