Learn how to use Scala with Play

Browse

- Table of contents

Contents

Select the version

- scala-0.9.1 ⬅
- scala-0.9

Search

Get help with google

Anorm, SQL data access with Play Scala

The Scala module includes a brand new data access layer called **Anorm** that uses plain SQL to make your database request and provides several API to parse and transform the resulting dataset.

> **Anorm is Not a Object Relational Mapper**

In the following documentation, we will use the MySQL world sample database.

If you want to enable it for your application, follow the MySQL website instruction, and enable it for your application by adding the following configuration line in your **conf/application.conf** file:

db=mysql:root@world

Overview

It can feel strange to fallback to plain old SQL to access an SQL Database these days. In particular for Java developers that are accustomed to use high level Object Relational Mapper like Hibernate to completely hide this aspect.

Now if we agree that these tools are almost required in Java, we think that they are not needed at all wehen you have the power of a higher level programming language like Scala, and in the contrary they will quickly become counter productive.

Using JDBC is a pain, but we provide a better API

We agree, using directly the JDBC API is tedious. Particularly in Java. You have to deal everywhere with checked exceptions and iterate over and over around the ResultSet to transform this raw dataset into your own data structure.

But we provide a simpler API for JDBC, using Scala you don't need to bother with exceptions, and transforming data is really easy with a functional language; in fact it is the point of the Play Scala SQL access layer to provide several API to effectively transform JDBC data into other Scala structures.

You don't need another DSL to access relational Database

SQL is already the best DSL to access relational Databases. We don't need to invent something new. Moreover the SQL syntax and features can differ from one database vendor to another.

If you try to abstract this point with another proprietary SQL like DSL you will have to deal with several 'dialects' dedicated for each vendor (like Hibernate ones), and limit yourself of using interesting features of a particular Database.

Sometimes we will provide you with prefilled SQL statements. But the idea is not to hide you the fact that we use SQL under the hood. Just save a bunch of characters to type for trivial queries, and you can always fallback to plain old SQL.

A type safe DSL to generate SQL is a mistake

Some argue that a type safe DSL is better since all your queries are checked by the compiler. Unfortunately the compiler check your queries based on a MetaModel definition that you often write yourself by 'mapping' your data structure to the database schema.

And there are no guarantees at all that this MetaModel is correct. Even if the compiler says that you code and your queries are correctly typed, it can still miserably fail at runtime because of a mismatch in your actual database definition.

Take Control of your SQL code

> Object Relationnal Mapper work well for trivial cases. But when you have to deal with complex schemas or existing databases, you will spend most of your time to fighting with your ORM to make it generate the SQL queries you want.
>
> Writing yourself SQL queries can be tedious for a simple 'Hello World' application, but for any real life application, you will eventually save time and simplify your code by taking the full control of your SQL code.

Now, let's see of to manage an SQL database with Play Scala.

Executing SQL requests

To start you need to learn how to execute SQL requests.

Well, import `play.db.anorm._`, and then simply use the `SQL` object to create queries.

```
import play.db.anorm._

val result:Boolean = SQL("Select 1").execute()
```

The `execute()` method returns a `Boolean` value indicating if the execution was succesful.

To execute an update query, you can use `executeUpdate()` that returns a `MayErr[IntegrityConstraintViolation,Int]` value:

```
val result = SQL("delete from City where id = 99").executeUpdate().fold(
    e => "Oops, there was an error" ,
    c => c + " rows were updated!"
)
```

Since Scala supports multiline String, feel free to use them for complex SQL statements:

```
var sqlQuery = SQL(
    """
        select * from Country c
        join CountryLanguage l on l.CountryCode = c.Code
        where c.code = 'FRA';
    """
)
```

If your SQL query needs dynamic parameters, you can declare placeholders like `{name}` in the query String, and assign them later to any value:

```
SQL(
    """
        select * from Country c
        join CountryLanguage l on l.CountryCode = c.Code
        where c.code = {countryCode};
    """
).on("countryCode" -> "FRA")
```

Another variant is to fill them by position:

```
SQL(
```

```
    """
        select * from Country c
        join CountryLanguage l on l.CountryCode = c.Code
        where c.code = {countryCode};
    """
).onParams("FRA")
```

Retrieving data using the Stream API

The first way to access data coming from a **Select** query, is to use the Stream API.

When you call `apply()` on any SQL statement, you will receive a lazy `Stream` of Row, where each row can be seen as a dictionary:

```
// Create an SQL query
val selectCountries = SQL("Select * from Country")

// Transform the resulting Stream[Row] as a List[(String,String)]
val countries = selectCountries().map(row =>
    row[String]("code") -> row[String]("name")
).toList
```

In the following example we will count the number of Country in the database. So the resultSet will be a single row with a single column:

```
// First retrieve the first row
val firstRow = SQL("Select count(*) as c from Country").apply().head

// Next get the content of the 'c' column as Long
val countryCount = firstRow[Long]("c")
```

Using Pattern Matching

You can also use Pattern Matching to match and extract the Row content. In this case the column name doesn't matter. Only the order and the type of the parameters is used to match.

The following example transform each row to the correct Scala type:

```
case class SmallCountry(name:String)
case class BigCountry(name:String)
case class France

val countries = SQL("Select name,population from Country")().collect {
    case Row("France", _) => France()
    case Row(name:String, pop:Int) if(pop > 1000000) => BigCountry(name)
    case Row(name:String, _) => SmallCountry(name)
}
```

Note that since `collect(…)` ignore the cases where the partial function isn't defined, it allow your code to safely ignore rows that you don't expect.

Dealing with Nullable columns

If a column can contain **Null** values in the database schema, you need to manipulate it as an

`Option` type.

For example, the **indepYear** of the **Country** table being nullable, you need to match it as `Option[Short]`:

```
SQL("Select name,indepYear from Country")().collect {
    case Row(name:String, Some(year:Short)) => name -> year
}
```

If you try to match this column as `Short` it won't be able to parse `Null` cases. If you try to retrieve the column content as `Short` directly from the dictionnary:

```
SQL("Select name,indepYear from Country")().map { row =>
    row[String]("name") -> row[Short]("indepYear")
}
```

It will produce an `UnexpectedNullableFound(COUNTRY.INDEPYEAR)` exception if it encounter a null value. So you need to map it properly to an `Option[Short]`, as:

```
SQL("Select name,indepYear from Country")().map { row =>
    row[String]("name") -> row[Option[Short]]("indepYear")
}
```

This rule is also true for the parser API we will just see.

Using the Parser combinator API

The Scala Parsers API provides generic parser combinators. Play Scala can use them to parse the result of any Select query.

First you need to import `play.db.anorm.SqlParser._`.

Use the `as(…)` method of the SQL statement to specify the parser you want to use. For example `scalar[Long]` is a simple parser that knows how to parse a single column row as `Long`:

```
val count:Long = SQL("select count(*) from Country").as(scalar[Long])
```

Let's write a more complicated parser:

`str("name") ~< int("population") *`, will parse the content of the **name** column as String, then the content of the **population** column as `Int`, and will repeat for each row. Here we use **~<** to combine several parsers that read the same row.

```
val populations:List[String~Int] = {
    SQL("select * from Country").as( str("name") ~< int("population") * )
}
```

As you see, the result type of this query is a `List[String~Int]`, so a list of country name and population items.

You can also, use `Symbol` and rewrite the same code as:

```
val populations:List[String~Int] = {
    SQL("select * from Country").as('name.of[String]~<'population.of[Int]*)
}
```

Or even as:

```
val populations:List[String~Int] = {
    SQL("select * from Country").as(
        get[String]("name") ~< get[Int]("population") *
    )
}
```

When you parse a **ResultSet** using `as(…)` it must consume all the input. If your parser doesn't consume all the available input, an error will be thrown. It avoids to have your parser fails silently.

If you want to parse only a small part of the input, you can use `parse(…)` instead of `as(…)`. However use it with caution, as it make it more difficult to detect errors in your code:

```
val onePopulation:String~Int = {
    SQL("select * from Country").parse(
        str("name") ~< int("population")
    )
}
```

Now let's try with a more complicated example. How to parse the result of the following query?

```
select c.name, c.code, l.language from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = 'FRA'
```

As this query uses a **join**, our parser will need to span several rows of the ResultSet to generate a single item. We will use the `spanM` combinator to construct this parser:

```
str("name") ~< spanM(by=str("code"), str("language"))
```

Now let's use this parser to create a function that gives us all languages spoken in a country:

```
case class SpokenLanguages(country:String, languages:Seq[String])

def spokenLanguages(countryCode:String):Option[SpokenLanguages] = {
    SQL(
        """
            select c.name, c.code, l.language from Country c
            join CountryLanguage l on l.CountryCode = c.Code
            where c.code = {code};
        """
    )
    .on("code" -> countryCode)
    .as(
        str("name") ~< spanM(by=str("code"), str("language")) ^^ {
            case country~languages => SpokenLanguages(country, languages)
        } ?
    )

}
```

Finally, let's complicate our example to separate the official language and the other ones:

```
case class SpokenLanguages(
    country:String,
```

```
    officialLanguage: Option[String],
    otherLanguages:Seq[String]
)

def spokenLanguages(countryCode:String):Option[SpokenLanguages] = {
    SQL(
        """
            select * from Country c
            join CountryLanguage l on l.CountryCode = c.Code
            where c.code = 'FRA';
        """
    ).as(
        str("name") ~< spanM(
            by=str("code"), str("language") ~< str("isOfficial")
        ) ^^ {
            case country~languages =>
                SpokenLanguages(
                    country,
                    languages.collect { case lang~"T" => lang } headOption,
                    languages.collect { case lang~"F" => lang }
                )
        } ?
    )

}
```

If you try this on the **world** sample database, you will get:

```
$ spokenLanguages("FRA")
> Some(
    SpokenLanguages(France,Some(French),List(
        Arabic, Italian, Portuguese, Spanish, Turkish
    ))
)
```

Adding some Magic[T]

Based on all these concepts, Play provides a **Magic** helper that will help you to write parsers. The idea is that if you define a `case class` that match a database table, Play Scala will generate a parser for you.

The `Magic` parsers need a convention to map you Scala structures to your database scheme. In this example we will use the default convention that map Scala case classes to Tables using exactly the class names as table name, and the field names as column names.

So before continuing, you need to import:

```
import play.db.anorm.defaults._
```

Let's try by defining a first `Country` case class that describes the **Country** table:

```
case class Country(
    code:Id[String], name:String, population:Int, headOfState:Option[String]
)
```

Note that we are not required to specify every existing table column in the case class. Only a subset is enough.

Now let's create an object that extends `Magic` to automatically get a parser of `Country`:

```
object Country extends Magic[Country]
```

If you want to break the convention here and use a different table name to for the `Country` case class, you can specify it:

```
object Country extends Magic[Country]().using("Countries")
```

And we can simply use **Country** as `Country` parser:

```
val countries:List[Country] = SQL("select * from Country").as(Country*)
```

`Magic` provides automatically a set of methods that can generate basic SQL queries:

```
val c:Long = Country.count().single()
val c:Long = Country.count("population > 1000000").single()
val c:List[Country] = Country.find().list()
val c:List[Country] = Country.find("population > 1000000").list()
val c:Option[Country] = Country.find("code = {c}").on("c" -> "FRA").first()
```

`Magic` also provides the `update` and `insert` methods. For example:

```
Country.update(Country(Id("FRA"), "France", 59225700, Some("Nicolas S.")))
```

Finally, let's write the missing `City` and `CountryLanguage` case classes, and make a more complex query:

```
case class Country(
    code:Id[String], name:String, population:Int, headOfState:Option[String]
)

case class City(
    id:Pk[Int], name: String
)

case class CountryLanguage(
    language:String, isOfficial:String
)

object Country extends Magic[Country]
object CountryLanguage extends Magic[CountryLanguage]
object City extends Magic[City]

val Some(country~languages~capital) = SQL(
    """
        select * from Country c
        join CountryLanguage l on l.CountryCode = c.Code
        join City v on v.id = c.capital
        where c.code = {code}
    """
)
.on("code" -> "FRA")
.as( Country.span( CountryLanguage * ) ~< City ? )
```

```
val countryName = country.name
val capitalName = capital.name
val headOfState = country.headOfState.getOrElse("No one?")

val officialLanguage = languages.collect {
                           case CountryLanguage(lang, "T") => lang
                       }.headOption.getOrElse("No language?")
```

Comments

Use this form to add corrections, additions and suggestions about the documentation on this page. Please ask questions on the play-framework group instead. Support requests, bug reports, and off-topic comments will be deleted without warning.

- Disqus
  - Login
  - About Disqus
- Like
- Dislike

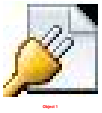
Glad you liked it. Would you like to share?

- Share
- No thanks

Sharing this page …

Thanks! Close

Add New Comment



Showing 7 comments

Sort by Popular now Best rating Newest first Oldest first  Subscribe by email

Subscribe by RSS

-  *Guest* 1 month ago

Anyone can point me to the api doc for Anorm? I couldn't find any details documentation of play.db.anorm._

Thanks

- Flag
1 person liked this.

-  *H Sf* 1 month ago

□□□ JPA。 □□□□□□□。 。 。 。 □□scala module □□□□□□ orm

- Flag

-  *Przemysław Pokrywka* 2 months ago

Guys, I'd like you to notice, that the real benefit of type safe DSLs is centralized metamodel (and not guarantee of its correctness). If schema changes, what would you like to do? To search for all places, where you have SQLs and correct it possibly by trial and error process? Or just to correct metamodel and have the compiler tell you all places, that need update?
Otherwise Anorm features several nice ideas, but the lack of type safety is too serious flaw for me to use it. Besides of making refactoring difficult, it violates DRY principle, because I have to repeat the type of a column in all places I use it (for example in firstRow[Long]("c")). You cannot also safely extract fragments of SQL statements to be

reused in other places when SQL lives in Strings.

Instead of Anorm I would use Squeryl / QueryDSL / ScalaQuery at data layer, even when using Play.

- Flag

 Wil Moore III and 8 more liked this

-
-
-

 *Alex Dean* 1 month ago in reply to Przemysław Pokrywka

-

I agree with you - the original author's rejection of type safety is nonsensical. Anorm is also useless for anybody who wants to release an open source project supporting more than one database technology. Disappointed it's being pushed as the standard Scala Play option.

- Flag
1 person liked this.
-
-
-

 *Guest* 1 month ago in reply to Przemysław Pokrywka

-

I share your concerns. I would appreciate if this chapter addressed this issue. Or at least linked to a discussion of it.

- Flag
-
-
-

 *eptx* 2 months ago

-

@OMAROMAN
I'm wondering the same thing: 1-1, 1-m, m-m, etc.

- Flag
-
-
-

 *OMAROMAN* 2 months ago

-

Hi,

Does anybody know how to implement a ManyToMany relationship using Anorm?

I have two models, User and Role, and I want to relate them using a join table, but I don't know how to indicate in the models the references just like in Hibernate/JPA

@Entity
@Table(name = "users")
public class User extends Model {
@ManyToMany
public List<role> roles;

...
}

@Entity
@Table(name = "roles")
public class Role extends Model {
@Required
public String name;
}

Here's my code in Scala:

EVOLUTIONS

CREATE TABLE users (
id bigint(20) NOT NULL AUTO_INCREMENT,
username varchar(20) NOT NULL,
email varchar(255) NOT NULL,
password_hash varchar(255) NOT NULL,
password_salt varchar(255) NOT NULL,
PRIMARY KEY (id)
);

CREATE TABLE roles (
id bigint(20) NOT NULL AUTO_INCREMENT,
name varchar(20) NOT NULL,
PRIMARY KEY (id)
);

CREATE TABLE roles_users (
role_id bigint(20) NOT NULL,
user_id bigint(20) NOT NULL,
FOREIGN KEY (role_id) REFERENCES roles(id),
FOREIGN KEY (user_id) REFERENCES users(id)
);

SCALA MODELS

case class User(
id: Pk[Long],
@Required username:String,
email:String,
password_hash:String,
password_salt:String,
)
object User extends Magic[User](Option("users"))

case class Role(
id: Pk[Long],
@Required name:String
)
object Role extends Magic[Role](Option("roles"))

Any help would be very apprecited.
Thanks ahead of time</role>

- Flag
  3 people liked this.
-

Learn how to use Scala with Play

Browse

- Table of contents
- Next: Data binding

Contents

Select the version

Search

Get help with google

Writing Play controllers in Scala

Play controllers are the most important part of any Play applications. A Play Scala application share the same concepts than a classical Play application but use a more functional way to describe actions.

Scala controllers are Objects

A Controller is a Scala singleton object, hosted by the `controllers` package, and subclassing `play.mvc.Controller`. In Scala you can declare as many controllers you want in the same file.

This a classical controller definition:

```
package controllers {
```

```
    import play._
    import play.mvc._

    object Users extends Controller {

        def show(id:Long) = Template("user" -> User.findById(id))

        def edit(id:Long, email:String) = {
            User.changeEmail(id, email)
            Action(show(id))
        }

    }

}
```

Because Scala provides the native notion of **Singleton objects** we don't need anymore to deal with Java static methods while keeping to ability to reference statically any action like `show(id)`.

Action methods return values

A Play controller usually uses imperative orders like `render(…)` or `forbidden()` to trigger the response generation. On the contrary an action methods written in Scala is seen as functions and must return a value. This value will be used by the framework to generate the HTTP response resulting of the request.

An action method can of course return several kind of values depending of the request (like for example a `Template` or an `Forbidden` value).

Here are listed the typical return types:

Ok

Returning the `Ok` value will generate an empty **200 OK** response.

```
def index = Ok
```

Html

Returning an `Html` value will generated a **200 OK** response filled with the HTML content. The response content type will be automatically set to **text/html**.

```
def index = Html("<h1>Hello world!</h1>")
```

You can also generate Html by calling a template.

Xml

Returning an `Xml` value will generated a **200 OK** response filled with the XML content. The response content type will be automatically set to **text/xml**.

```
def index = Xml(<message>Hello world!</message>)
```

Text

Returning an `Text` value will generated a **200 OK** response filled with the text content. The response content type will be automatically set to **text/plain**.

```
def index = Text("Hello world!")
```

Json

Returning an `Json` value will generated a **200 OK** response filled with the text content. The response content type will be automatically set to **application/json**.

```
def index = Json("{message: 'Hello world'}")
```

You can also try to pass any Scala object and Play will try to serialize it to JSON:

```
def index = Json(users)
```

However currently the JSON serialization mechanism comes from Java and can not work as exepected with complex Scala structures.

A workaround is to use a Scala dedicated JSON serialization library, for example Lift JSON, and use it as `Json(JsonAST.render(users))`

Created

Returning the `Created` value will generate an empty **201 Created** response.

```
def index = Created
```

Accepted

Returning the `Accepted` value will generate an empty **202 Accepted** response.

```
def index = Accepted
```

NoContent

Returning the `NoContent` value will generate an empty **204 No Content** response.

```
def index = NoContent
```

Action

If an action method return a `Action` value, Play will redirect the Browser to the corresponding action, using the action method arguments to properly resolve the proper URL.

```
def index = Action(show(3))
```

Note that here `show(3)` is a **by-name** parameter, and the corresponding methid will not been invoked. Play will resolve this call as an URL (typically something like **users/3**), and will issue an HTTP redirect to this URL. The action will then be invoked in a new request context.

In a Java controller you achieve the same result by calling directly the corresponding action method. Using Scala **call by name** concept allow to keep the compiler checked and typesafe redirection without any language hack.

Redirect

Returning the `Redirect` value will generate an empty **301 Moved Permanently** response.

```
def index = Redirect("http://www.google.com")
```

You can optionnaly specify a second argument to switch between **301** and **302** response status code.

```
def index = Redirect("http://www.google.com", false)
```

NotModified

Returning the `NotModified` value will generate an empty **304 Not Modified** response.

```
def index = NotModified
```

You can also specify an **ETag** to the response:

```
def index = NotModified("123456")
```

BadRequest

Returning the `BadRequest` value will generate an empty **400 Bad Request** response.

```
def index = BadRequest
```

Unauthorized

Returning the `Unauthorized` value will generate an empty **401 Unauthorized** response.

```
def index = Unauthorized
```

You can optionnaly specify a realm name:

```
def index = Unauthorized("Administration area")
```

Forbidden

Returning the `Forbidden` value will generate an empty **403 Forbidden** response.

```
def index = Forbidden
```

You can optionnaly specify an error message:

```
def index = Forbidden("Unsufficient permissions")
```

NotFound

Returning the `NotFound` value will generate an empty **404 Not Found** response.

```
def index = NotFound
```

You can optionnaly specify a resource name:

```
def index = NotFound("Article not found")
```

Or use a more classical HTTP method, resource Path combination:

```
def index = NotFound("GET", "/toto")
```

Error

Returning the `Error` value will generate an empty **500 Internal Server Error** response.

```
def index = Error
```

You can optionnaly specify an error message:

```
def index = Error("Oops…")
```

Or specify a more specific error code:

```
def index = Error(503, "Not ready yet…")
```

Return type inference

You can also directly use the inferred return type to send the action result. For example using a String:

```
def index = "<h1>Hello world</h1>"
```

Or you can even use the built-in XML support to write XHTML in a literal way:

```
def index = <h1>Hello world</h1>
```

If the return type looks like a binary stream, play will automatically render the response as binary. So generating a captcha image using the built-in Captcha helper can be written as:

```
def index = Images.captcha
```

Controller interceptors

Controller interceptors work almost the same way than for Java controller. You simply have to annotate any controller method with the corresponding interceptor annotation:

```
@Before def logRequests {
    println("New request…")
}
```

You see that here, the `logRequests` method does not return any value. So the request execution will continue by invoking the next interceptors and eventually the action method.

But you can also write some interceptor that return a value:

```
@Before def protectActions = {
    Forbidden
}
```

Here the execution will stop, and the `Forbidden` value will be used to generate the HTTP response.

If you want to continue the request execution, just make your interceptor return `Continue`:

```
@Before def protectActions = {
    session("isAdmin") match {
        case Some("yes") => Continue
        case _ => Forbidden("Restricted to administrators")
    }
}
```

Mixing controllers using Traits

Scala Traits can be used to compose controller more effeciently by mixing several aspects. You can define both action methods and interceptors in a controller Trait.

For example the following **Secure** trait add a seucrity interceptor to any controller applying the Trait:

```
trait Secure {
```

```
    self:Controller =>

    @Before checkSecurity = {
        session("username") match {
            case Some(username) => renderArgs += "user" -> User(username)
                                   Continue
            case None => Action(Authentication.login)
        }
    }

    def connectedUser = renderArgs("user").get

}
```

Note that here we use the `self:Controller =>` notation to indicate that this Trait can only be mixed with a `Controller` type.

And you can use it to create a secured controller:

```
object Application extends Controller with Secure {

    def index = <h1>Hello {connectedUser.name}!</h1>

}
```

There is also small differences about Data binding

Comments

Use this form to add corrections, additions and suggestions about the documentation on this page. Please ask questions on the play-framework group instead. Support requests, bug reports, and off-topic comments will be deleted without warning.

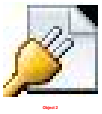
Glad you liked it. Would you like to share?

- No thanks

Sharing this page …

Thanks! Close

Add New Comment



Showing 4 comments

Sort by Popular now Best rating Newest first Oldest first Subscribe by email Subscribe by RSS

- *Alejandro Paz* 1 month ago

I found a typo in the first blue box: "Scala without limiting the expressivness of" -> "Scala without limiting the expressiveness of"

- Flag

- *William Lee* 1 month ago

Is there a way to return my own object for a particular content type?

- Flag

- *Dr Karl* 2 months ago

There is a typo in the Secure trait code snippet. There should be a new line after @Before, and most important a "def" is lacking before "checkSecurity"

- Flag
  1 person liked this.
-
-
-

 *Louis Gueye* 2 months ago

-

Hi there,
Great work. Scala really needs something less ugly and more reachable in concept than lift ...
By the way you can correct this typo :
"and the corresponding methid will not been" --> "and the corresponding method will not been"

Louis

- Flag
-
-

Scala templates

Play Scala comes with a new and really poweful Scala based template engine. The design of this new template engine is really inspired by ASP.NET Razor, especially:

Compact, Expressive, and Fluid: Minimizes the number of characters and keystrokes required in a file, and enables a fast, fluid coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote server blocks within your HTML. The parser is smart enough to infer this from your code. This enables a really compact and expressive syntax which is clean, fast and fun to type.

Easy to Learn: Enables you to quickly be productive with a minimum of concepts. You use all your existing Scala language and HTML skills.

Is not a new language: We consciously chose not to create a new language. Instead we wanted to enable developers to use their existing Scala language skills, and deliver a template markup syntax that enables an awesome HTML construction workflow with your language of choice.

Works with any Text Editor: Razor doesn’t require a specific tool and enables you to be productive in any plain old text editor.

Overview

A Play Scala template is a simple text file text file, that contains small blocks of Scala code. It can generate any text-based format (HTML, XML, CSV, etc.).

It's particularely designed to feel comfortable to those used to working with HTML, allowing Web designers to work with.

They are compiled as standard Scala functions, following a simple naming convention:

> If you create a **views/Application/index.scala.html** template file, it will generate a `views.Application.html.index` function.

Here is for example, a classic template content:

```
@(customer:models.Customer, orders:Seq[models.Order])

<h1>Welcome @customer.name!</h1>

@if(orders) {

    <h2>Here is a list of your current orders:</h2>

    <ul>
    @orders.map { order =>
        <li>@order.title</li>
    }
    </ul>

} else {

    <h2>You don't have any order yet...</h2>

}
```

And you can easily use it from any Scala code:

```
val page:play.template.Html = views.Application.html.index(
    customer, orders
)
```

Syntax: the magic '@' character

The Scala template uses **'@'** as single special character. Each time this character is encountered, it indicates the begining of a Scala statement. It does not require you to explicitly close the code-block, and will infer it from your code:

```
Hello @customer.name!
      ^^^^^^^^^^^^^^
        Scala code
```

Because the template engine will automatically detect the end of your code block by analysing your code, it only allow for simple statements. If you want to insert a multi-token statement, just make it more explicit using brackets:

```
Hello @(customer.firstName + customer.lastName)!
```

```
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    Scala Code
```

You can also use curly bracket, like in plain Scala code, to write a multi-statements block:

```
Hello @{val name = customer.firstName + customer.lastName; name}!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                             Scala Code
```

Because **'@'** is the only special character, if you want to escape it, just use **'@@'**

Template parameters

Because a template is a function, it needs parameters. Template parameters must be declared on the first template line:

```
@(customer:models.Customer, orders:Seq[models.Order])
```

You can also use **default values** for parameters:

```
@(title:String = "Home")
```

Or even several parameter groups:

```
@(title:String)(body: => Html)
```

And even implicit parameters:

```
@(title:String)(body: => Html)(implicit session:play.mvc.Scope.Session)
```

Note that all parameter type names must be fully qualified.

Looping

You can use the Scala `for comprehension`, is a pretty standard way. Just note that the template compiler will just add a `yield` keyword before your block:

```
<ul>
@for(p <- products) {
    <li>@p.name ($@p.price)</li>
}
</ul>
```

But as you probably know, here the `for comprehension` is just syntaxic sugar for a classic map:

```
<ul>
@products.map { p =>
    <li>@p.name ($@p.price)</li>
}
</ul>
```

If-Blocks

Nothing special here. Just use the `if` instruction from Scala:

```
@if(items.isEmpty) {
    <h1>Nothing to display</h1>
} else {
    <h1>@items.size items!</h1>
}
```

Pattern matching

You can also use pattern matching in templates:

```
@connected match {

    case Admin(name) => {
        <span class="admin">Connected as admin (@name)</span>
    }

    case User(name) => {
        <span>Connected as @name</span>
    }

}
```

Declaring reusable blocks

You can create reusable code block (or sub template):

```
@display(product:models.Product) = {
    @product.name ($@product.price)
}

<ul>
@products.map { p =>
    @display(product = p)
}
</ul>
```

Note that you can also declare reusable pure Scala blocks:

```
@title(text:String) = @{
    text.split(' ').map(_.capitalize).mkString(" ")
}

<h1>@title("hello world")</h1>
```

Import statements

You can import whatever you want at the begining of your template (or of a sub template):

```
@(customer:models.Customer, orders:Seq[models.Order])
```

```
@import utils._

…
```

Composing templates (tags, layouts, includes, etc.)

Templates being simple functions you can compose them in any way you want. Below are a few examples of other common scenarios:

Layout

Let's declare a **views/main.scala.html** template that will act as main layout:

```
@(title:String)(content: => Html)

<h1>@title</h1>

<hr>

<div id="main">
    @content
</div>

<hr>

<div id="footer">
    ...
</div>
```

As you see this template takes 2 parameters: a title and an HTML block.

Now we can use it from another **views/Application/index.scala.html** template:

```
@main(title = "Home") {

    <h1>Home page</h1>

}
```

Tags

Let's write a simple **views/tags/notice.scala.html** tag that display an HTML notice:

```
@(level:String = "error")(body: (String) => Html)

@level match {

    case "success" => {
        <p class="success">
            @body("green")
        </p>
    }

    case "warning" => {
        <p class="warning">
```

```
            @body("orange")
        </p>
    }

    case "error" => {
        <p class="error">
            @body("red")
        </p>
    }

}
```

And let's use it from any template:

```
@import views.tags.html._

@notice("error") { color =>
    Oops, something is <span style="color:@color">wrong</span>
}
```

Includes

Nothing special, you can just call any other templates:

```
<h1>Home</h1>

<div id="side">
    @views.common.html.sideBar()
</div>
```

Comments

Use this form to add corrections, additions and suggestions about the documentation on this page. Please ask questions on the play-framework group instead. Support requests, bug reports, and off-topic comments will be deleted without warning.

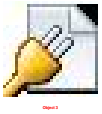
Glad you liked it. Would you like to share?

- Share
- No thanks

Sharing this page …

Thanks! Close

Add New Comment



Showing 7 comments

Sort by Popular now Best rating Newest first Oldest first Subscribe by email Subscribe by RSS

- 

  Guest 3 weeks ago
- How to use the "route functions" or "actions"?
  http://www.playframework.org/d...
- Flag
-
-
- 

  AlexSerov51 1 month ago
- Is it an interpreter?
  Razor is of course the latest from Microsoft. But still not perfect.
  Looks like it there are no components at all.
  How is it better that Velocity not even to mention HybridJava
- Flag
-
-

-  □□ □ 1 month ago
- would it support jade? it is easy and dry too. similar to ruby slim, http://slim-lang.com/
- Flag
-
-
-  *Zhaopuming* 1 month ago
- Would the template support Jade/Scamel style code in the future ? I found those to be really really "Compact, Expressive, and Fluid"
- Flag
-  □□ □ liked this
-
-
-  *felipe peko* 1 month ago
- It would be nice to explain the use of package objects so the user can give views some default functions/imports. I found it very useful, and it helps with DRY principle.
- Flag
-
-
-  *Ponny* 1 month ago
- Does it escape HTML? If so, how do you prevent it? If not, how do you force it?

- Flag
  1 person liked this.
-
-
-

 *Diego Varese* 1 month ago

-

This @ syntax is very similar to the one used by the Razor view engine in ASP.NET MVC. It would be great if it were available for java code as well.

-

Scala templates

Play Scala comes with a new and really poweful Scala based template engine. The design of this new template engine is really inspired by ASP.NET Razor, especially:

Compact, Expressive, and Fluid: Minimizes the number of characters and keystrokes required in a file, and enables a fast, fluid coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote server blocks within your HTML. The parser is smart enough to infer this from your code. This enables a really compact and expressive syntax which is clean, fast and fun to type.

Easy to Learn: Enables you to quickly be productive with a minimum of concepts. You use all your existing Scala language and HTML skills.

Is not a new language: We consciously chose not to create a new language. Instead we wanted to enable developers to use their existing Scala language skills, and deliver a template markup syntax that enables an awesome HTML construction workflow with your language of choice.

Works with any Text Editor: Razor doesn't require a specific tool and enables you to be productive in any plain old text editor.

Overview

A Play Scala template is a simple text file text file, that contains small blocks of Scala code. It can generate any text-based format (HTML, XML, CSV, etc.).

It's particularely designed to feel comfortable to those used to working with HTML, allowing Web designers to work with.

They are compiled as standard Scala functions, following a simple naming convention:

> If you create a **views/Application/index.scala.html** template file, it will generate a `views.Application.html.index` function.

Here is for example, a classic template content:

```
@(customer:models.Customer, orders:Seq[models.Order])

<h1>Welcome @customer.name!</h1>

@if(orders) {

    <h2>Here is a list of your current orders:</h2>

    <ul>
    @orders.map { order =>
        <li>@order.title</li>
    }
    </ul>

} else {

    <h2>You don't have any order yet...</h2>

}
```

And you can easily use it from any Scala code:

```
val page:play.template.Html = views.Application.html.index(
    customer, orders
)
```

Syntax: the magic '@' character

The Scala template uses **'@'** as single special character. Each time this character is encountered, it indicates the begining of a Scala statement. It does not require you to explicitly close the code-block, and will infer it from your code:

```
Hello @customer.name!
      ^^^^^^^^^^^^^^
        Scala code
```

Because the template engine will automatically detect the end of your code block by analysing your code, it only allow for simple statements. If you want to insert a multi-token statement, just make it more explicit using brackets:

```
Hello @(customer.firstName + customer.lastName)!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    Scala Code
```

You can also use curly bracket, like in plain Scala code, to write a multi-statements block:

```
Hello @{val name = customer.firstName + customer.lastName; name}!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                               Scala Code
```

Because **'@'** is the only special character, if you want to escape it, just use **'@@'**

Template parameters

Because a template is a function, it needs parameters. Template parameters must be declared on the first template line:

```
@(customer:models.Customer, orders:Seq[models.Order])
```

You can also use **default values** for parameters:

```
@(title:String = "Home")
```

Or even several parameter groups:

```
@(title:String)(body: => Html)
```

And even implicit parameters:

```
@(title:String)(body: => Html)(implicit session:play.mvc.Scope.Session)
```

Note that all parameter type names must be fully qualified.

Looping

You can use the Scala `for comprehension`, is a pretty standard way. Just note that the template compiler will just add a `yield` keyword before your block:

```
<ul>
@for(p <- products) {
    <li>@p.name ($@p.price)</li>
}
</ul>
```

But as you probably know, here the `for comprehension` is just syntaxic sugar for a classic map:

```
<ul>
@products.map { p =>
    <li>@p.name ($@p.price)</li>
}
</ul>
```

If-Blocks

Nothing special here. Just use the `if` instruction from Scala:

```
@if(items.isEmpty) {
    <h1>Nothing to display</h1>
} else {
    <h1>@items.size items!</h1>
}
```

Pattern matching

You can also use pattern matching in templates:

```
@connected match {

    case Admin(name) => {
        <span class="admin">Connected as admin (@name)</span>
    }

    case User(name) => {
        <span>Connected as @name</span>
    }

}
```

Declaring reusable blocks

You can create reusable code block (or sub template):

```
@display(product:models.Product) = {
    @product.name ($@product.price)
}

<ul>
@products.map { p =>
    @display(product = p)
}
</ul>
```

Note that you can also declare reusable pure Scala blocks:

```
@title(text:String) = @{
    text.split(' ').map(_.capitalize).mkString(" ")
}

<h1>@title("hello world")</h1>
```

Import statements

You can import whatever you want at the begining of your template (or of a sub template):

```
@(customer:models.Customer, orders:Seq[models.Order])

@import utils._

…
```

Composing templates (tags, layouts, includes, etc.)

Templates being simple functions you can compose them in any way you want. Below are a few examples of other common scenarios:

Layout

Let’s declare a **views/main.scala.html** template that will act as main layout:

```
@(title:String)(content: => Html)

<h1>@title</h1>

<hr>

<div id="main">
    @content
</div>

<hr>

<div id="footer">
    ...
</div>
```

As you see this template takes 2 parameters: a title and an HTML block.

Now we can use it from another **views/Application/index.scala.html** template:

```
@main(title = "Home") {

    <h1>Home page</h1>

}
```

Tags

Let's write a simple **views/tags/notice.scala.html** tag that display an HTML notice:

```
@(level:String = "error")(body: (String) => Html)

@level match {

    case "success" => {
        <p class="success">
            @body("green")
        </p>
    }

    case "warning" => {
        <p class="warning">
            @body("orange")
        </p>
    }

    case "error" => {
        <p class="error">
            @body("red")
        </p>
    }

}
```

And let's use it from any template:

```
@import views.tags.html._
```

```
@notice("error") { color =>
    Oops, something is <span style="color:@color">wrong</span>
}
```

Includes

Nothing special, you can just call any other templates:

```
<h1>Home</h1>

<div id="side">
    @views.common.html.sideBar()
</div>
```

Comments

Use this form to add corrections, additions and suggestions about the documentation on this page. Please ask questions on the play-framework group instead. Support requests, bug reports, and off-topic comments will be deleted without warning.

- Disqus
  - Login
  - About Disqus
- Like
- Dislike

Glad you liked it. Would you like to share?

- Share
- No thanks

Sharing this page …

Thanks! Close

Add New Comment

Showing 7 comments

Sort by Popular now Best rating Newest first Oldest first  Subscribe by email

 Subscribe by RSS

-  *Guest* ★ 3 weeks ago
- How to use the "route functions" or "actions"?
  http://www.playframework.org/d...
- Flag
-
-
-  *AlexSerov51* ★ 1 month ago
- Is it an interpreter?
  Razor is of course the latest from Microsoft. But still not perfect.
  Looks like it there are no components at all.
  How is it better that Velocity not even to mention HybridJava
- Flag
-
-
-  □□ □ ★ 1 month ago
- would it support jade? it is easy and dry too. similar to ruby slim, http://slim-lang.com/

- Flag
-
-
-

 *Zhaopuming* 1 month ago

-

Would the template support Jade/Scamel style code in the future ? I found those to be really really "Compact, Expressive, and Fluid"

- Flag

 liked this

-
-
-

 *felipe peko* 1 month ago

-

It would be nice to explain the use of package objects so the user can give views some default functions/imports. I found it very useful, and it helps with DRY principle.

- Flag
-
-
-

 *Ponny* 1 month ago

-

Does it escape HTML? If so, how do you prevent it? If not, how do you force it?

- Flag

1 person liked this.

-
-
-

 *Diego Varese* 1 month ago

- This @ syntax is very similar to the one used by the Razor view engine in ASP.NET MVC. It would be great if it were available for java code as well.
- Flag
  7 people liked this.
-
-