The play framework is fully stateless and only Request/Response oriented. All HTTP Requests follow the same path:

* An HTTP Request is received by the framework.
* The Router component tries to find the most specific route able to accept this request. The corresponding action method is then invoked.
* The application code is executed.
* If a complex view needs to be generated, a template file is rendered.
* The result of the action method (HTTP Response code, Content) is then written as an HTTP Response.

The framework compiles the Java source code at runtime and only keep compiled classes in a bytecode cache under the tmp directory. The main executables artifacts in a play application are the .java source files, not the compiled classes.

There are three standard packages[controllers, model, view] in the app directory, one for each layer of the MVC architectural pattern. You can of course add your own packages like for example an utils packages.

The conf directory : contains all configuration files for the application. There are two required configurations files:

application.conf: the main configuration file for the application. It contains standard configuration options.

routes: the routes definition file.

The lib directory

This directory contains all standard Java libraries needed by your application. These are automatically added to the Java Classpath.

The public directory

Resources stored in the public directory are static assets and are served directly by the Web server.

This directory is split into three standard sub-directories: for images, CSS stylesheets and Javascript files.

By default the /public directory is mapped to the /public URL path.

**Controllers**

* Controllers represent the thin layer between HTTP and business logic.
* They are called by the router.
* Requested resource is mapped to a controller method specified in the conf/routes file.

public class Application extends Controller {

public static void showUser(String id) {

render();

}

}

This render() method is defined in the controller class and tells the controller to start the rendering phase. A template is looked up and rendered.

As Play also follows the convention over configuration pattern, a default template location is assumed, which follows an easy naming scheme:

./app/views/${controller}/{method}.html

In the case of showing a user it would be:

./app/views/Application/showUser.html

Requisite:

* Java class work as controller is to have it extend from play.mvc.Controller.
* All your actions[Methods] have to be static and public in order to be executed as controller methods.

**Router:**

GET / controllers.Application.index()

* GET /new controllers.Application.newIndex()
* URL - <http://localhost:9000/new>
* method newIndex(Long id) of application class is called.
* URL - <http://localhost:9000/new/> will not work.
* GET /clients**/:id** controllers.Application.clients(**id: Long**)
* URL - [http://localhost:9000/clients/**10**](http://localhost:9000/clients/10)
* 10 is passed as argument in method clients(Long id) of application class.
* GET /files/\*name controllers.Application.several(name)
* <http://localhost:9000/files/1234/4567/Manish>
* “1234/4567/Manish” is passed as argument in method several(String name) of application class.
* \* of Files/**\***name is for “1234/4567/Manish” and
* **name** is the argument passed in method several(String name).
* /files/\*name controllers.Application.several(name). in route name must match.

**application.conf** [conf\]

This is the file where basic information such as

* + database connection strings,
  + log levels,
  + modules to enable additional functionality,
  + supported application languages, or
  + the setting of the application mode

is configured.

every configuration option follows the scheme:

# Some comment

key = value

Adding and configuring modules in order to enhance the basic functionality of Play is part of the application.conf, and each module requires enabling it via defining its path:

module.foo=${play.path}/modules/foo.

NOTE:

From play 1.2 modules are not configured via this mechanism, but via the new dependencies.yml file. You can still configure modules this way, but this is deprecated from then on.

Another important setup is the log level of your application when using log4j-

application.log=DEBUG

application.conf file looks like a properties file, it really is not because of its UTF-8 encoding.

render() - This method is defined in the controller class and tells the controller to start the rendering phase.

A template is looked up and rendered.

As Play also follows the convention over configuration pattern, a default template location is assumed, which follows an easy naming

Best-Practices:

* Controller layer should be as thin as possible.
* This layer is **not** for business logic but merely for validation in order to ensure the model layer will only get valid data.

Thread safety

Static methods in a controller are not threadsafe!. True

However, the Controller is bytecode enhanced in order to make certain calls threadsafe. If you are interested in knowing more, check the class play.classloading.enhancers.ControllerEnhancer.

**Model**

Any model you define should go into the models package[in the app/models 1directory].

import play.db.jpa.Model;

@Entity

**public** **class** User **extends** Model {

**public** String login;

@Required @Email

**public** String email;

}

Key-Observations:

* All fields in the example code are public
  + Warning bells for Java developer - violation of encapsulation
  + Play generates getters and setters (and are injected at runtime) while preserving encapsulation. More to come on this
* The @Entity annotation marks this class as a managed JPA entity.
* By default, the table name is ‘User’ in database as class name is same.
  + If you need to use different table name for this class annotate the User class with
  + @Table(name=“blog\_user”).

The model[User] class inherits from Model.

This is absolutely essential, because it allows you to use the so-called ActiveRecord pattern for querying of data.

you can use the save() method to persist the object to the database.

**Test**