



PLAY2 FRAMEWORK

For Scala Developer



DECEMBER 13, 2015
CONSOLIDIATE BY SCOTT HUANG
Personal Hobby

Contents Index

Actions, Controllers and Results.....	29
What is an Action?	29
Building an Action.....	29
Controllers are action generators.....	30
Simple results	30
Redirects are simple results too.....	31
TODO dummy page	32
HTTP routing	32
The built-in HTTP router.....	32
Dependency Injection.....	32
The routes file syntax	33
The HTTP method.....	33
The URI pattern.....	33
Static path.....	33
Dynamic parts.....	33
Dynamic parts spanning several /.....	34
Dynamic parts with custom regular expressions.....	34
Call to the Action generator method	34
Parameter types	35
Parameters with fixed values	35
Parameters with default values.....	35
Optional parameters.....	35
Routing priority.....	35
Reverse routing	35
Manipulating Results.....	36
Changing the default Content-Type	36
Manipulating HTTP headers.....	37
Setting and discarding cookies	37
Changing the charset for text based HTTP responses.....	38

Session and Flash scopes	38
How it is different in Play	39
Storing data in the Session	39
Reading a Session value	40
Discarding the whole session	40
Flash scope	40
Body parsers	41
What is a Body Parser?	41
More about Actions	42
Default body parser: AnyContent	42
Specifying a body parser	43
Combining body parsers	43
Max content length	44
Action composition	44
Custom action builders	45
Composing actions	45
More complicated actions	46
Different request types	47
Authentication	48
Adding information to requests	48
Validating requests	49
Putting it all together	49
Content negotiation	49
Language	49
Content	50
Request extractors	50
Handling errors	51
Supplying a custom error handler	51
Extending the default error handler	52
Handling asynchronous results	53
Make controllers asynchronous	53
Creating non-blocking actions	53
How to create a <code>Future [Result]</code>	53

Returning futures.....	54
Actions are asynchronous by default.....	55
Handling time-outs	55
Streaming HTTP responses	55
Standard responses and <code>Content-Length</code> header.....	56
Sending large amounts of data.....	56
Serving files	57
Chunked responses.....	58
Comet sockets.....	59
Using chunked responses to create Comet sockets	59
Using the <code>play.api.libs.Comet</code> helper.....	60
The forever iframe technique.....	61
WebSockets	61
Handling WebSockets.....	61
Handling WebSockets with actors	62
Detecting when a WebSocket has closed.....	62
Closing a WebSocket.....	63
Rejecting a WebSocket.....	63
Handling different types of messages	63
Handling WebSockets with iteratees.....	64
The template engine.....	66
A type safe template engine based on Scala.....	66
Overview.....	67
Syntax: the magic '@' character	68
Template parameters.....	68
Iterating	69
If-blocks.....	69
Declaring reusable blocks.....	69
Declaring reusable values.....	70
Import statements.....	70
Comments	70
Escaping.....	71
String interpolation.....	71

Scala templates common use cases.....	71
Layout	71
Tags (they are just functions, right?).....	73
Includes.....	73
moreScripts and moreStyles equivalents.....	74
Adding support for a custom format to the template engine.....	75
Overview of the templating process.....	75
Implement a format	76
Associate a file extension to the format.....	76
Tell Play how to make an HTTP result from a template result type	77
Handling form submission.....	77
Overview.....	77
Imports	77
Form Basics.....	78
Defining a form	78
Defining constraints on the form	79
Defining ad-hoc constraints.....	80
Validating a form in an Action	81
Showing forms in a view template	82
Displaying errors in a view template.....	83
Mapping with tuples.....	83
Mapping with single.....	84
Fill values	84
Nested values.....	84
Repeated values	85
Optional values.....	85
Default values.....	85
Ignored values	86
Putting it all together	86
Protecting against Cross Site Request Forgery.....	87
Play's CSRF protection	88
Applying a global CSRF filter	88
Getting the current token.....	89

Adding a CSRF token to the session.....	90
Applying CSRF filtering on a per action basis.....	90
CSRF configuration options.....	91
Using Custom Validations.....	92
Custom Field Constructors	92
Writing your own field constructor.....	93
JSON basics.....	94
The Play JSON library.....	94
JsValue	94
Json	95
JsPath	95
Converting to a JsValue	95
Using string parsing.....	95
Using class construction.....	95
Using Writes converters	96
Traversing a JsValue structure	98
Simple path \	98
Recursive path \\	98
Index lookup (for JsArrays)	98
Converting from a JsValue.....	98
Using String utilities	98
Using JsValue.as/asOpt.....	99
Using validation	99
JsValue to a model.....	100
JSON with HTTP	101
Serving a list of entities in JSON.....	101
Creating a new entity instance in JSON.....	102
Summary	104
JSON Reads/Writes/Format Combinators.....	104
JsPath	105
Reads.....	106
Path Reads.....	106
Complex Reads	106

Validation with Reads.....	107
Putting it all together	107
Writes.....	108
Recursive Types.....	109
Format.....	109
Creating Format from Reads and Writes.....	109
Creating Format using combinators	110
JSON transformers	110
Introducing JSON <i>coast-to-coast</i> design	110
Are we doomed to convert JSON to OO?.....	110
Is OO conversion really the default use case?.....	111
New tech players change the way of manipulating JSON.....	111
JSON <i>coast-to-coast</i> design.....	111
JSON transformers are <code>Reads[T <: JsValue]</code>	112
Use <code>JsValue.transform</code> instead of <code>JsValue.validate</code>	112
The details	112
Case 1: Pick JSON value in JsPath	113
Pick value as JsValue	113
Pick value as Type	113
Case 2: Pick branch following <code>JsPath</code>	114
Pick branch as <code>JsValue</code>	114
Case 3: Copy a value from input JsPath into a new JsPath.....	114
Case 4: Copy full input Json & update a branch.....	115
Case 5: Put a given value in a new branch.....	116
Case 6: Prune a branch from input JSON.....	117
More complicated cases	117
Case 7: Pick a branch and update its content in 2 places	118
Case 8: Pick a branch and prune a sub-branch.....	119
What about combinators?	119
JSON Macro Inception.....	121
Writing a default case class Reads/Writes/Format is so boring!.....	121
Let's be minimalist.....	122
JSON Inception.....	123

Code Equivalence	123
Inception equation.....	123
Json inception is Scala 2.10 Macros.....	124
Writes[T] & Format[T].....	125
Writes[T].....	126
Format[T].....	126
Special patterns	126
Known limitations.....	126
Handling and serving XML requests	127
Handling an XML request	127
Serving an XML response	128
Handling file upload.....	128
Uploading files in a form using multipart/form-data.....	128
Direct file upload.....	129
Writing your own body parser	129
Accessing an SQL database	130
Configuring JDBC connection pools.....	130
H2 database engine connection properties.....	131
SQLite database engine connection properties	132
PostgreSQL database engine connection properties	132
MySQL database engine connection properties.....	132
How to configure several data sources	132
Configuring the JDBC Driver.....	132
Accessing the JDBC datasource	132
Obtaining a JDBC connection.....	133
Selecting and configuring the connection pool.....	134
Testing	134
Enabling Play database evolutions.....	134
Using Play Slick.....	134
Getting Help.....	135
About this release.....	135
Setup.....	135
Support for Play database evolutions.....	135

JDBC driver dependency.....	136
Database Configuration	136
Usage.....	139
DatabaseConfig via Dependency Injection	139
DatabaseConfig via Global Lookup.....	139
Running a database query in a Controller.....	139
Configuring the connection pool	140
Play Slick Migration Guide	140
Build changes	140
Removed H2 database dependency.....	140
Evolutions support in a separate module	140
Database configuration	141
Automatic Slick driver detection.....	141
<code>DBAction</code> and <code>DBSessionRequest</code> were removed.....	142
Thread Pool.....	142
<code>Profile</code> was removed.....	143
<code>Database</code> was removed.....	143
<code>Config</code> was removed	143
<code>SlickPlayIteratees</code> was removed.....	143
DDL support was removed	143
Play Slick Advanced Topics.....	144
Connection Pool.....	144
Thread Pool.....	145
Play Slick FAQ.....	145
What version should I use?.....	145
<code>play.db.pool</code> is ignored.....	145
Changing the connection pool used by Slick	145
A binding to <code>play.api.db.DBApi</code> was already configured.....	146
Play throws <code>java.lang.ClassNotFoundException: org.h2.tools.Server</code>	146
Anorm, simple SQL data access	147
Overview.....	147
Add Anorm to your project	148
Executing SQL queries	149

SQL queries using String Interpolation	151
Streaming results.....	151
Multi-value support.....	153
Batch update.....	154
Edge cases.....	154
Using Pattern Matching.....	156
Using for-comprehension.....	156
Retrieving data along with execution context.....	156
Working with optional/nullable values.....	157
Using the Parser API.....	158
Getting a single result.....	158
Getting a single optional result.....	158
Getting a more complex result.....	158
A more complicated example	160
JDBC mappings	161
Column parsers	162
Parameters	165
Integrating with other database libraries.....	169
Integrating with ScalaQuery	169
Exposing the datasource through JNDI.....	170
The Play cache API.....	170
Importing the Cache API.....	171
Accessing the Cache API	171
Accessing different caches	172
Caching HTTP responses.....	172
Control caching.....	173
Custom implementations	173
The Play WS API.....	174
Making a Request	174
Request with authentication.....	175
Request with follow redirects	175
Request with query parameters.....	175
Request with additional headers.....	175

Request with virtual host	175
Request with timeout.....	175
Submitting form data	175
Submitting JSON data.....	175
Submitting XML data.....	176
Processing the Response.....	176
Processing a response as JSON.....	176
Processing a response as XML.....	176
Processing large responses.....	177
Common Patterns and Use Cases.....	178
Chaining WS calls	178
Using in a controller.....	179
Using WSClient	179
Configuring WS	180
Configuring WS with SSL.....	181
Configuring Timeouts	181
Configuring AsyncHttpClientConfig	181
OpenID Support in Play	181
The OpenID flow in a nutshell.....	182
Usage.....	182
OpenID in Play	182
Extended Attributes	183
OAuth.....	184
Usage.....	184
Required Information.....	184
Authentication Flow	184
Example.....	185
Integrating with Akka	186
The application actor system	186
Writing actors.....	186
Creating and using actors	187
Asking things of actors	187
Dependency injecting actors.....	188

Dependency injecting child actors	189
Configuration	191
Changing configuration prefix	191
Built-in actor system name.....	191
Scheduling asynchronous tasks	191
Using your own Actor system	192
Messages and internationalization	192
Specifying languages supported by your application.....	193
Externalizing messages	193
Messages format.....	193
Notes on apostrophes	194
Retrieving supported language from an HTTP request.....	194
Testing your application	194
Advanced testing.....	194
Testing your application with ScalaTest	195
Overview.....	195
Using ScalaTest + Play	195
Matchers	196
Mockito.....	196
Unit Testing Models.....	197
Unit Testing Controllers.....	198
Unit Testing EssentialAction	200
Writing functional tests with ScalaTest	200
FakeApplication.....	201
Testing with a server	202
Testing with a web browser	203
Running the same tests in multiple browsers	205
PlaySpec.....	208
When different tests need different fixtures.....	209
Testing a template.....	213
Testing a controller	214
Testing the router	214
Testing a model.....	214

Testing WS calls.....	215
Testing your application with specs2.....	215
Overview.....	215
Using specs2.....	215
Matchers.....	216
Mockito.....	217
Unit Testing Models.....	218
Unit Testing Controllers.....	219
Unit Testing EssentialAction.....	219
Writing functional tests with specs2.....	220
FakeApplication.....	220
WithApplication.....	221
WithServer.....	221
WithBrowser.....	222
PlaySpecification.....	223
Testing a view template	223
Testing a controller.....	223
Testing the router	224
Testing a model.....	224
Testing with Guice.....	224
GuiceApplicationBuilder	224
Environment	224
Configuration	225
Bindings and Modules.....	225
GuiceInjectorBuilder	226
Overriding bindings in a functional test.....	226
Testing with databases	227
Using a database	228
Allowing Play to manage the database for you.....	229
Using an in-memory database.....	230
Applying evolutions.....	231
Custom evolutions.....	231
Allowing Play to manage evolutions.....	232

Testing web service clients	233
Test against the actual web service.....	233
Test against a test instance of the web service.....	233
Mock the http client.....	234
Mock the web service.....	234
Testing a GitHub client.....	234
Returning files.....	236
Extracting setup code.....	237
The Logging API.....	238
Logging architecture.....	238
Using Loggers	239
Configuration	242
Handling data streams reactively.....	242
Iteratees.....	242
Some important types in the <code>Iteratee</code> definition:.....	243
Some primitive iteratees:.....	243
Folding input:	245
Handling data streams reactively.....	246
Enumerators.....	246
Enumerators à la carte.....	249
Handling data streams reactively	249
The realm of Enumerates.....	249
Introduction to Play HTTP API.....	253
What is EssentialAction?.....	253
Bottom Line.....	254
Filters	254
Filters vs action composition.....	255
A simple logging filter	255
Using filters	256
Where do filters fit in?	256
More powerful filters	257
HTTP Request Handlers.....	258
Implementing a custom request handler	258

Extending the default request handler.....	259
Configuring the http request handler	260
Performance notes.....	260
Runtime Dependency Injection.....	260
Declaring dependencies	261
Dependency injecting controllers.....	261
Injected routes generator	261
Injected actions.....	262
Component lifecycle.....	262
Singletons	262
Stopping/cleaning up.....	262
Providing custom bindings	263
Play applications.....	263
Play libraries	266
Excluding modules	266
Advanced: Extending the GuiceApplicationLoader.....	266
Compile Time Dependency Injection.....	267
Current application	268
Application entry point	268
Providing a router	269
Using other components.....	271
String Interpolating Routing DSL	271
Javascript Routing.....	273
Generating a Javascript router.....	274
Embedded router.....	274
Router resource.....	274
Using the router	275
jQuery ajax method support.....	275
Writing Plugins.....	276
Implementing plugins	276
Accessing plugins.....	277
Actor example.....	277
Embedding a Play server in your application	278

The Build System	279
Understanding sbt	279
Play application directory structure	280
The <code>/build.sbt</code> file	280
Using scala for building	281
The <code>/project</code> directory	281
Play plugin for sbt (<code>/project/plugins.sbt</code>)	281
About SBT Settings	282
About sbt settings	282
Default settings for Java applications	282
Default settings for Scala applications	282
Managing library dependencies	283
Unmanaged dependencies	283
Managed dependencies	283
Getting the right Scala version with <code>%%</code>	284
Resolvers	284
Working with sub-projects	284
Adding a simple library sub-project	284
Sharing common variables and code	286
Splitting your web application into several parts	287
Splitting the route file	287
Consider the following build configuration	287
Project structure	288
Assets and controller classes should be all defined in the <code>controllers.admin</code> package	288
Reverse routing in <code>admin</code>	289
Through the browser	289
Play enhancer	290
Motivation	290
Drawbacks	290
Setting up	291
Operation	291
Configuration	291
Aggregating reverse routers	292

Improving Compilation Times	293
Use subprojects/modularize.....	293
Annotate return types of public methods.....	293
Avoid large cycles between source files.....	293
Minimize inheritance.....	293
SBT Cookbook	294
Hook actions around <code>play run</code>	294
Add compiler options.....	295
Add additional asset directory	295
Disable documentation.....	296
Configure ivy logging level	296
Fork and parallel execution in test	296
Debugging your build	296
Debugging dependencies	297
Debugging settings.....	297
The show command.....	298
The inspect command.....	298
The inspect tree command	299
Debugging incremental compilation	299
Working with public assets.....	300
The public/ folder	300
WebJars.....	300
How are public assets packaged?.....	301
The Assets controller.....	301
Reverse routing for public assets.....	302
Reverse routing and fingerprinting for public assets	302
Etag support	303
Gzip support	303
Additional <code>Cache-Control</code> directive.....	303
Managed assets.....	303
Using CoffeeScript	304
Layout	305
Enablement and Configuration	306

Using LESS CSS	306
Working with partial LESS source files.....	307
Layout	308
Using LESS with Bootstrap.....	308
Enablement and Configuration	308
Using JSHint.....	309
Check JavaScript sanity.....	309
Enablement and Configuration	309
RequireJS.....	309
Deployment	310
Enablement and Configuration	310
Configuration file syntax and features	311
Specifying an alternative configuration file	311
Using with Akka.....	311
Using with the <code>run</code> command	311
Extra <code>devSettings</code>	312
HTTP server settings in <code>application.conf</code>	312
HOCON Syntax.....	312
Unchanged from JSON.....	312
Comments	312
Omit root braces.....	312
Key-value separator	312
Commas	312
Duplicate keys	313
Paths as keys	314
Substitutions	315
Includes.....	316
Include syntax.....	316
Include semantics: merging	317
Include semantics: substitution	318
Include semantics: missing files.....	318
Include semantics: locating resources.....	319
Duration format.....	320

Size in bytes format.....	320
Conventional override by system properties.....	321
The Application Secret.....	321
Best practices	321
Environment variables.....	322
Production configuration file	322
Generating an application secret	322
Updating the application secret in application.conf	323
Configuring the JDBC pool.	323
Special URLs.....	323
Reference	324
Understanding Play thread pools.....	328
Knowing when you are blocking	329
Play's thread pools	329
Using the default thread pool	330
Configuring the Play default thread pool.....	330
Using other thread pools	331
Class loaders and thread locals.....	331
Application class loader.....	332
Java thread locals.....	332
Best practices	333
Pure asynchronous	333
Highly synchronous	333
Many specific thread pools.....	334
Few specific thread pools.....	335
Configuring logging	335
Default configuration.....	335
Custom configuration	337
Using a configuration file from project source	337
Using an external configuration file.....	337
Examples	337
Akka logging configuration.....	338
Configuring gzip encoding.....	339

Enabling the gzip filter	339
Configuring the gzip filter.....	340
Controlling which responses are gzipped.....	340
Configuring Security Headers.....	340
Enabling the security headers filter.....	341
Configuring the security headers	341
Cross-Origin Resource Sharing.....	342
Enabling the CORS filter.....	342
Configuring the CORS filter.....	343
Configuring WS SSL.....	343
Table of Contents.....	344
Further Reading.....	344
Quick Start to WS SSL.....	344
Connecting to a Remote Server over HTTPS.....	344
Obtain the Root CA Certificate.....	345
Point the trust manager at the PEM file.....	346
Generating X.509 Certificates.....	346
X.509 Certificates	346
Using Keytool	347
Generating a random password	347
Server Configuration	347
Generating a server CA	347
Generating example.com certificates	348
Configuring example.com certificates in Nginx	349
Client Configuration.....	350
Configuring a Trust Store	351
Configure Client Authentication	351
Certificate Management Tools.....	354
Certificate Settings.....	354
Secure.....	354
Compatible.....	354
Further Reading.....	354
Configuring Trust Stores and Key Stores.....	354

Configuring a Trust Manager	355
Configuring a Key Manager	355
Configuring a Store.....	356
Debugging	356
Further Reading.....	356
Configuring Protocols.....	357
Defining the default protocol	357
Debugging	357
Configuring Cipher Suites	358
Configuring Enabled Ciphers.....	358
Recommendation: increase the DHE key size	359
Recommendation: Use Ciphers with Perfect Forward Secrecy.....	359
Disabling Weak Ciphers and Weak Key Sizes Globally.....	359
Debugging	360
Configuring Certificate Validation.....	360
Disabling Certificates with Weak Signature Algorithms	361
Disabling Certificates With Weak Key Sizes	361
Disabling Weak Certificates Globally.....	362
Debugging Certificate Validation.....	362
Further Reading.....	362
Configuring Certificate Revocation	363
Debugging	363
Further Reading.....	364
Configuring Hostname Verification	364
Modifying the Hostname Verifier.....	364
Debugging	365
Further Reading.....	365
Example Configurations.....	365
Connecting to an internal web service.....	365
Connecting to an internal web service with client authentication.....	365
Connecting to several external web services.....	366
Both Private and Public Servers.....	366
Using the Default SSLContext.....	367

Debugging	367
Debugging SSL Connections	367
Verbose Debugging	368
Dynamic Debugging.....	369
Further reading	369
Loose Options	369
We understand.....	369
Please read this before turning anything off!.....	369
Man in the Middle attacks are well known.....	369
Man in the Middle attacks are common.....	370
Attackers have a suite of tools that automatically exploit flaws	370
Security is increasingly important and public.....	370
Ethernet / Password protected WiFi does not provide a meaningful level of security.....	370
Companies have been sued for inadequate security.....	370
Correctly configured HTTPS clients are important.....	370
Mitigation.....	370
Loose Options	371
Disabling Certificate Verification.....	371
Disabling Weak Ciphers Checking	371
Disabling Hostname Verification.....	372
Disabled Protocols.....	372
Testing SSL.....	372
Unit Testing.....	372
Integration Testing.....	372
Adversarial Testing.....	375
Testing Certificate Verification.....	375
Testing Weak Cipher Suites	375
Testing Certificate Validation.....	375
Testing Hostname Verification	375
H2 database	375
Target databases	376
Prevent in memory DB reset.....	376
Caveats.....	376

H2 Browser.....	377
H2 Documentation	377
Managing database evolutions.....	377
Enable evolutions.....	377
Evolutions scripts.....	377
Evolutions configuration	379
Synchronizing concurrent changes	380
Inconsistent states.....	382
Transactional DDL.....	384
Evolution storage and limitations	384
Deploying your application.....	384
The application secret	384
Using the dist task	385
The Native Packager.....	386
Build a server distribution	386
Including additional files in your distribution.....	387
Play PID Configuration.....	387
Publishing to a Maven (or Ivy) repository	388
Running a production server in place	388
Running a test instance.....	389
Using the SBT assembly plugin.....	390
Additional configuration	390
General configuration.....	390
Specifying an alternate configuration file	391
Overriding configuration with system properties	392
Using environment variables.....	392
Server configuration options	392
Logging configuration.....	395
Bundling a custom logback configuration file with your application.....	395
Using <code>-Dlogger.resource</code>	395
Using <code>-Dlogger.file</code>	395
Using <code>-Dlogger.url</code>	395
JVM configuration.....	396

Setting up a front end HTTP server	396
Set up with lighttpd	396
Set up with nginx	397
Set up with Apache	398
Advanced proxy settings.....	399
Apache as a front proxy to allow transparent upgrade of your application	399
Configure trusted proxies	400
Configuring HTTPS.....	401
Providing configuration.....	401
SSL Certificates.....	401
SSL Certificates from a keystore	401
SSL Certificates from a custom SSL Engine.....	402
Turning HTTP off.....	403
Production usage of HTTPS.....	403
Deploying a Play application to a cloud service	403
Deploying to Heroku.....	404
Deploying to a remote Git repository	404
Store your application in git.....	404
Create a new application on Heroku	404
Deploy your application.....	404
Check that your application has been deployed	405
Deploying with the sbt-heroku plugin.....	406
Adding the plugin.....	406
Deploying with the plugin	407
Connecting to a database	407
Further learning resources.....	408
Deploying to CloudFoundry / AppFog	409
Prerequisites.....	409
Build your Application	409
Deploy your Application	409
Working With Services	410
Auto-Reconfiguration.....	410
Connecting to Cloud Foundry Services.....	410

Opting out of Auto-Reconfiguration	411
Deploying to Clever Cloud	411
Create a new application on Clever Cloud	411
Deploy your application.....	411
Check the deployment of your application	412
[Optional] Configure your application.....	412
Connecting to a database	412
Further information	412
Deploying to Boxfuse and AWS	413
Prerequisites.....	413
Build your Application.....	413
Deploy your Application	413
Further learning resources.....	414
Akka HTTP server backend(<i>experimental</i>).....	414
Known issues	415
Usage.....	415
Manually selecting the Akka HTTP server	415
Verifying that the Akka HTTP server is running	415
Configuring the Akka HTTP server.....	415
Reactive Streams integration (<i>experimental</i>)	416
Known issues	416
Usage.....	417
Building Play from source	417
Grab the source.....	417
Build the documentation.....	418
Run tests.....	418
Use in projects.....	418
Using Code in Eclipse	419
Artifact repositories.....	419
Typesafe repository.....	419
Accessing snapshots	419
Issues tracker	419
Reporting bugs	419

Guidelines for writing Play documentation	420
Markdown	420
Links.....	420
Code samples	421
Scala	422
Java.....	422
Scala Templates.....	422
Routes files	423
SBT code	423
Other code.....	423
Testing the docs	423
Code samples from external Play modules.....	424
Translating the Play Documentation.....	424
Prerequisites.....	425
Setting up a translation.....	425
Translating documentation.....	425
Dealing with code samples	426
Validating the documentation.....	427
Translation report	427
Deploying documentation to playframework.com.....	427
Specifying the documentation version.....	428
Working with Git	428
Git remotes	428
Branches.....	428
Squashing commits.....	429
Responding to reviews/build breakages.....	430
Starting over	430
A word on changing history.....	431
3rd Party Tools	431
Continuous Integration.....	431
Profiling	432
Introducing Play 2.....	432
Built for asynchronous programming.....	433

Focused on type safety	433
Native support for Java and Scala.....	434
Powerful build system	435
Datastore and model integration.....	436
Play User Groups.....	436
New York	436
Berlin	436
Cologne	436
Scala User Group Köln / Bonn.....	436
Buenos Aires	436
Stockholm.....	437
Belgium.....	437
Japan	437
Republic of Korea.....	437
New Delhi - INDIA	437
What's new in Play 2.4.....	437
Dependency Injection.....	437
Motivation.....	438
Approach	438
Testing	438
Embedding Play	439
Aggregated reverse routers	439
Java 8 support.....	439
Maven/sbt standard layout.....	440
Anorm	440
Ebean.....	441
HikariCP.....	441
WS	441
Experimental Features.....	441
Akka HTTP support.....	441
Reactive Streams Support.....	441
Play Modules	442
Airbrake.io notifier.....	442

Amazon SES module (Scala).....	442
Amazon S3 module (Scala).....	442
Amf module (Scala).....	442
Authentication and Authorization module (Scala)	442
Authenticity Token module.....	443
ClojureScript Plugin.....	443
Cloudfront module (Scala)	443
Currency Converter (Java).....	443
Deadbolt 2 Plugin.....	443
DDSL Plugin - Dynamic Distributed Service Locator.....	443
Dust Plugin	443
Google Closure Template Plugin.....	444
Elasticsearch	444
Ember.js.....	444
funcy - Page Driven Functional Tests (Java)	444
FolderMessages plugin.....	444
Flyway plugin.....	444
Geolocation (Java)	444
Google's HTML Compressor (Java and Scala).....	445
Groovy Templates plugin	445
Groovy Templates plugin - gt-engine-play2	445
Guice Plugin (Java and Scala).....	445
HTML5 Tags module (Java and Scala).....	445
InputValidator (Scala).....	446
JackRabbit Plugin (Java and Scala).....	446
Japid module	446
JsMessages.....	446
JSON minification Plugin.....	446
JSONP filter	446
Lessc Plugin	446
Liquibase Module	446
Manual Dependency Injection Plugin (Java and Scala)	447
Memcached Plugin.....	447

Messages Compiler Plugin (Scala)	447
MongoDB Jackson Mapper Plugin (Java).....	447
MongoDB Jongo Plugin (Java).....	447
MongoDB Morphia Plugin (Java).....	447
MongoDB Salat, Casbah Plugin (Scala)	448
Mountable routing.....	448
Mustache (Java,Scala).....	448
Native Packaging Module	448
NINA (Scala).....	448
Origami: OrientDB O/G Mapper (Java and Scala).....	448
PDF module (Java)	448
Play! Authenticate (Java)	449
play2-sprites	449
Play-Bootstrap3 (Java and Scala)	449
play-jaxrs (Java)	449
Play-pac4j (Java and Scala)	449
Play PlovPlugin	449
Play-Slick	449
Pusher.....	450
Play Dok.....	450
Qunit (Java)	450
Redis Plugin (Java and Scala).....	450
Swaggerkit (Scala)	450
Emailer Plugin (Java and Scala).....	450
Roy Compiled Asset Plugin (Ray)	450
Sass Plugin.....	450
ScalikeJDBC Plugin (Scala).....	451
SecureSocial (Java and Scala).....	451
Silhouette (Scala)	451
Sitemap Generator (Java).....	451
Snapshot Plugin (Java and Scala)	451
socket.io.play (scala only, pre-alpha).....	451
Stateless client authentication (Scala).....	452

Statsd Plugin (Java and Scala).....	452
Stylus Plugin	452
TinkerPop Frames O/G Mapper Plugins (Java).....	452
Typesafe util Plugin (Scala)	452
Typesafe SbtGoodies Plugin	452
TypeScript Plugin	452
WAR Module	453
XForward module.....	453
XWiki Rendering module (Scala).....	453
Thymeleaf module (Scala).....	453

Actions, Controllers and Results

What is an Action?

Most of the requests received by a Play application are handled by an `Action`.

A `play.api.mvc.Action` is basically a `(play.api.mvc.Request => play.api.mvc.Result)` function that handles a request and generates a result to be sent to the client.

```
val echo = Action { request =>
  Ok("Got request [" + request + "]")
}
```

An action returns a `play.api.mvc.Result` value, representing the HTTP response to send to the web client. In this example `Ok` constructs a **200 OK** response containing a `text/plain` response body.

Building an Action

The `play.api.mvc.Action` companion object offers several helper methods to construct an `Action` value.

The first simplest one just takes as argument an expression block returning a `Result`:

```
Action {
  Ok("Hello world")
```

```
}
```

This is the simplest way to create an Action, but we don't get a reference to the incoming request. It is often useful to access the HTTP request calling this Action.

So there is another Action builder that takes as an argument a function `Request =>`

```
Result:
```

```
Action { request =>
  Ok("Got request [" + request + "]")
}
```

It is often useful to mark the `request` parameter as `implicit` so it can be implicitly used by other APIs that need it:

```
Action { implicit request =>
  Ok("Got request [" + request + "]")
}
```

The last way of creating an Action value is to specify an additional `BodyParser` argument:

```
Action(parse.json) { implicit request =>
  Ok("Got request [" + request + "]")
}
```

Body parsers will be covered later in this manual. For now you just need to know that the other methods of creating Action values use a default **Any content body parser**.

Controllers are action generators

A `Controller` is nothing more than a singleton object that generates `Action` values.

The simplest use case for defining an action generator is a method with no parameters that returns an `Action` value :

```
package controllers
```

```
import play.api.mvc._

class Application extends Controller {

  def index = Action {
    Ok("It works!")
  }
}
```

Of course, the action generator method can have parameters, and these parameters can be captured by the `Action` closure:

```
def hello(name: String) = Action {
  Ok("Hello " + name)
}
```

Simple results

For now we are just interested in simple results: An HTTP result with a status code, a set of HTTP headers and a body to be sent to the web client.

These results are defined by `play.api.mvc.Result`:

```
def index = Action {  
    Result(  
        header = ResponseHeader(200, Map(CONTENT_TYPE -> "text/plain")),  
        body = Enumerator("Hello world!".getBytes())  
    )  
}
```

Of course there are several helpers available to create common results such as the `Ok` result in the sample above:

```
def index = Action {  
    Ok("Hello world!")  
}
```

This produces exactly the same result as before.

Here are several examples to create various results:

```
val ok = Ok("Hello world!")  
val notFound = NotFound  
val pageNotFound = NotFound(<h1>Page not found</h1>)  
val badRequest = BadRequest(views.html.form(formWithErrors))  
val oops = InternalServerError("Oops")  
val anyStatus = Status(488)("Strange response type")
```

All of these helpers can be found in the `play.api.mvc.Results` trait and companion object.

Redirects are simple results too

Redirecting the browser to a new URL is just another kind of simple result. However, these result types don't take a response body.

There are several helpers available to create redirect results:

```
def index = Action {  
    Redirect("/user/home")  
}
```

The default is to use a `303 SEE_OTHER` response type, but you can also set a more specific status code if you need one:

```
def index = Action {  
    Redirect("/user/home", MOVED_PERMANENTLY)  
}
```

TODO dummy page

You can use an empty `Action` implementation defined as `TODO`: the result is a standard 'Not implemented yet' result page:

```
def index(name:String) = TODO
```

[Next: HTTP Routing](#)

HTTP routing

The built-in HTTP router

The router is the component in charge of translating each incoming HTTP request to an Action.

An HTTP request is seen as an event by the MVC framework. This event contains two major pieces of information:

- the request path (e.g. `/clients/1542`, `/photos/list`), including the query string
- the HTTP method (e.g. `GET`, `POST`, ...).

Routes are defined in the `conf/routes` file, which is compiled. This means that you'll see route errors directly in your browser:

Dependency Injection

Play supports generating two types of routers, one is a dependency injected router, the other is a static router. The default is the static router, but if you created a new Play application using the Play seed Activator templates, your project will include the following configuration in `build.sbt` telling it to use the injected router:

```
routesGenerator := InjectedRoutesGenerator
```

The code samples in Play's documentation assumes that you are using the injected routes generator. If you are not using this, you can trivially adapt the code samples for the static routes generator, either by prefixing the controller invocation part of the route with an `@` symbol, or by declaring each of your controllers as an `object` rather than a `class`.

The routes file syntax

`conf/routes` is the configuration file used by the router. This file lists all of the routes needed by the application. Each route consists of an HTTP method and URI pattern, both associated with a call to an `Action` generator.

Let's see what a route definition looks like:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Each route starts with the HTTP method, followed by the URI pattern. The last element is the call definition.

You can also add comments to the route file, with the `#` character.

`# Display a client.`

```
GET /clients/:id controllers.Clients.show(id: Long)
```

The HTTP method

The HTTP method can be any of the valid methods supported by HTTP (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`).

The URI pattern

The URI pattern defines the route's request path. Parts of the request path can be dynamic.

Static path

For example, to exactly match incoming `GET /clients/all` requests, you can define this route:

```
GET /clients/all controllers.Clients.list()
```

Dynamic parts

If you want to define a route that retrieves a client by ID, you'll need to add a dynamic part:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Note that a URI pattern may have more than one dynamic part.

The default matching strategy for a dynamic part is defined by the regular expression `[^/]+`, meaning that any dynamic part defined as `:id` will match exactly one URI part.

Dynamic parts spanning several /

If you want a dynamic part to capture more than one URI path segment, separated by forward slashes, you can define a dynamic part using the `*id` syntax, which uses the `.+` regular expression:

```
GET /files/*name controllers.Application.download(name)
```

Here for a request like `GET /files/images/logo.png`, the `name` dynamic part will capture the `images/logo.png` value.

Dynamic parts with custom regular expressions

You can also define your own regular expression for the dynamic part, using the `$id<regex>` syntax:

```
GET /items/$id<[0-9]+> controllers.Items.show(id: Long)
```

Call to the Action generator method

The last part of a route definition is the call. This part must define a valid call to a method returning a `play.api.mvc.Action` value, which will typically be a controller action method.

If the method does not define any parameters, just give the fully-qualified method name:

```
GET / controllers.Application.homePage()
```

If the action method defines some parameters, all these parameter values will be searched for in the request URI, either extracted from the URI path itself, or from the query string.

```
# Extract the page parameter from the path.  
GET /:page controllers.Application.show(page)
```

Or:

```
# Extract the page parameter from the query string.  
GET / controllers.Application.show(page)  
Here is the corresponding, show method definition in  
the controllers.Application controller:
```

```
def show(page: String) = Action {  
    loadContentFromDatabase(page).map { htmlContent =>  
        Ok(htmlContent).as("text/html")  
    }.getOrElse(NotFound)  
}
```

Parameter types

For parameters of type `String`, typing the parameter is optional. If you want Play to transform the incoming parameter into a specific Scala type, you can explicitly type the parameter:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

And do the same on the corresponding `show` method definition in the `controllers.Clients` controller:

```
def show(id: Long) = Action {
  Client.findById(id).map { client =>
    Ok(views.html.Clients.display(client))
  }.getOrElse(NotFound)
}
```

Parameters with fixed values

Sometimes you'll want to use a fixed value for a parameter:

```
# Extract the page parameter from the path, or fix the value for /
GET / controllers.Application.show(page = "home")
GET /:page controllers.Application.show(page)
```

Parameters with default values

You can also provide a default value that will be used if no value is found in the incoming request:

```
# Pagination links, like /clients?page=3
GET /clients controllers.Clients.list(page: Int ?= 1)
```

Optional parameters

You can also specify an optional parameter that does not need to be present in all requests:

```
# The version parameter is optional. E.g. /api/list-all?version=3.0
GET /api/list-all controllers.Api.list(version: Option[String])
```

Routing priority

Many routes can match the same request. If there is a conflict, the first route (in declaration order) is used.

Reverse routing

The router can also be used to generate a URL from within a Scala call. This makes it possible to centralize all your URI patterns in a single configuration file, so you can be more confident when refactoring your application.

For each controller used in the routes file, the router will generate a ‘reverse controller’ in the `routes` package, having the same action methods, with the same signature, but returning a `play.api.mvc.Call` instead of a `play.api.mvc.Action`. The `play.api.mvc.Call` defines an HTTP call, and provides both the HTTP method and the URI.

For example, if you create a controller like:

```
package controllers

import play.api._
import play.api.mvc._

class Application extends Controller {

    def hello(name: String) = Action {
        Ok("Hello " + name + "!")
    }
}
```

And if you map it in the `conf/routes` file:

```
# Hello action
GET /hello/:name    controllers.Application.hello(name)
You can then reverse the URL to the hello action method, by using
the controllers.routes.Application reverse controller:
// Redirect to /hello/Bob
def helloBob = Action {
    Redirect(routes.Application.hello("Bob"))
}
```

Next: Manipulating results

Manipulating Results

Changing the default Content-Type

The result content type is automatically inferred from the Scala value that you specify as the response body.

For example:

```
val textResult = Ok("Hello World!")
```

Will automatically set the `Content-Type` header to `text/plain`, while:

```
val xmlResult = Ok(<message>Hello World!</message>)
```

will set the Content-Type header to `application/xml`.

Tip: this is done via the `play.api.http.ContentTypeOf` type class.

This is pretty useful, but sometimes you want to change it. Just use the `as(newContentType)` method on a result to create a new similar result with a different `Content-Type` header:

```
val htmlResult = Ok(<h1>Hello World!</h1>).as("text/html")
```

or even better, using:

```
val htmlResult2 = Ok(<h1>Hello World!</h1>).as(HTML)
```

Note: The benefit of using `HTML` instead of the `"text/html"` is that the charset will be automatically handled for you and the actual Content-Type header will be set to `text/html; charset=utf-8`. We will see that in a bit.

Manipulating HTTP headers

You can also add (or update) any HTTP header to the result:

```
val result = Ok("Hello World!").withHeaders(  
  CACHE_CONTROL -> "max-age=3600",  
  ETAG -> "xx")
```

Note that setting an HTTP header will automatically discard the previous value if it was existing in the original result.

Setting and discarding cookies

Cookies are just a special form of HTTP headers but we provide a set of helpers to make it easier.

You can easily add a Cookie to the HTTP response using:

```
val result = Ok("Hello world").withCookies(  
  Cookie("theme", "blue"))
```

Also, to discard a Cookie previously stored on the Web browser:

```
val result2 = result.discardCookies(DiscardingCookie("theme"))
```

You can also set and remove cookies as part of the same response:

```
val result3 = result.withCookies(Cookie("theme", "blue")).discardCookies(DiscardingCookie("skin"))
```

Changing the charset for text based HTTP responses

For text based HTTP response it is very important to handle the charset correctly. Play handles that for you and uses `utf-8` by default (see [why to use utf-8](#)).

The charset is used to both convert the text response to the corresponding bytes to send over the network socket, and to update the `Content-Type` header with the proper `; charset=xxx` extension.

The charset is handled automatically via the `play.api.mvc.Codec` type class. Just import an implicit instance of `play.api.mvc.Codec` in the current scope to change the charset that will be used by all operations:

```
class Application extends Controller {
```

```
    implicit val myCustomCharset = Codec.javaSupported("iso-8859-1")
```

```
    def index = Action {
        Ok(<h1>Hello World!</h1>).as(HTML)
    }
```

Here, because there is an implicit charset value in the scope, it will be used by both the `Ok(...)` method to convert the XML message into `ISO-8859-1` encoded bytes and to generate the `text/html; charset=iso-8859-1` Content-Type header.

Now if you are wondering how the `HTML` method works, here it is how it is defined:

```
def HTML(implicit codec: Codec) = {
    "text/html; charset=" + codec.charset
}
```

You can do the same in your API if you need to handle the charset in a generic way.

Next: [Session and Flash scopes](#)

Session and Flash scopes

How it is different in Play

If you have to keep data across multiple HTTP requests, you can save them in the Session or Flash scopes. Data stored in the Session are available during the whole user Session, and data stored in the Flash scope are available to the next request **only**.

It's important to understand that Session and Flash data are not stored by the server but are added to each subsequent HTTP request, using the cookie mechanism. This means that the data size is very limited (up to 4 KB) and that you can only store string values. The default name for the cookie is `PLAY_SESSION`. This can be changed by configuring the key `session.cookieName` in `application.conf`.

If the name of the cookie is changed, the earlier cookie can be discarded using the same methods mentioned in [Setting and discarding cookies](#).

Of course, cookie values are signed with a secret key so the client can't modify the cookie data (or it will be invalidated).

The Play Session is not intended to be used as a cache. If you need to cache some data related to a specific Session, you can use the Play built-in cache mechanism and store a unique ID in the user Session to keep them related to a specific user.

By default, there is no technical timeout for the Session. It expires when the user closes the web browser. If you need a functional timeout for a specific application, just store a timestamp into the user Session and use it however your application needs (e.g. for a maximum session duration, maximum inactivity duration, etc.). You can also set the maximum age of the session cookie by configuring the key `session.maxAge` (in milliseconds) in `application.conf`.

Storing data in the Session

As the Session is just a Cookie, it is also just an HTTP header. You can manipulate the session data the same way you manipulate other results properties:

```
Ok("Welcome!").withSession(  
    "connected" -> "user@gmail.com")
```

Note that this will replace the whole session. If you need to add an element to an existing Session, just add an element to the incoming session, and specify that as new session:

```
Ok("Hello World!").withSession(  
    request.session + ("saidHello" -> "yes"))
```

You can remove any value from the incoming session the same way:

```
Ok("Theme reset!").withSession(  
    request.session - "theme")
```

Reading a Session value

You can retrieve the incoming Session from the HTTP request:

```
def index = Action { request =>  
    request.session.get("connected").map { user =>  
        Ok("Hello " + user)  
    }.getOrElse {  
        Unauthorized("Oops, you are not connected")  
    }  
}
```

Discarding the whole session

There is special operation that discards the whole session:

```
Ok("Bye").withNewSession
```

Flash scope

The Flash scope works exactly like the Session, but with two differences:

- data are kept for only one request
- the Flash cookie is not signed, making it possible for the user to modify it.

Important: The Flash scope should only be used to transport success/error messages on simple non-Ajax applications. As the data are just kept for the next request and because there are no guarantees to ensure the request order in a complex Web application, the Flash scope is subject to race conditions.

Here are a few examples using the Flash scope:

```
def index = Action { implicit request =>  
    Ok {  
        request.flash.get("success").getOrElse("Welcome!")  
    }  
}  
  
def save = Action {  
    Redirect("/home").flashing(  
        "success" -> "The item has been created")  
}
```

To retrieve the Flash scope value in your view, add an implicit Flash parameter:

```
@()(implicit flash: Flash)  
...  
@flash.get("success").getOrElse("Welcome!")  
...
```

And in your Action, specify an `implicit request =>` as shown below:

```
def index = Action { implicit request =>  
  Ok(views.html.index())  
}
```

An implicit Flash will be provided to the view based on the implicit request.

If the error '*could not find implicit value for parameter flash: play.api.mvc.Flash*' is raised then this is because your Action didn't have an implicit request in scope.

Next: [Body parsers](#)

Body parsers

What is a Body Parser?

An HTTP PUT or POST request contains a body. This body can use any format, specified in the `Content-Type` request header. In Play, a **body parser** transforms this request body into a Scala value.

However the request body for an HTTP request can be very large and a **body parser** can't just wait and load the whole data set into memory before parsing it. A `BodyParser[A]` is basically an `Iteratee[Array[Byte], A]`, meaning that it receives chunks of bytes (as long as the web browser uploads some data) and computes a value of type `A` as result.

Let's consider some examples.

- A **text** body parser could accumulate chunks of bytes into a String, and give the computed String as result (`Iteratee[Array[Byte], String]`).
- A **file** body parser could store each chunk of bytes into a local file, and give a reference to the `java.io.File` as result (`Iteratee[Array[Byte], File]`).
- A **s3** body parser could push each chunk of bytes to Amazon S3 and give a the S3 object id as result (`Iteratee[Array[Byte], S3ObjectId]`).

Additionally a **body parser** has access to the HTTP request headers before it starts parsing the request body, and has the opportunity to run some precondition checks. For example, a body parser can check that some HTTP headers are properly set, or that the user trying to upload a large file has the permission to do so.

Note: That's why a body parser is not really an `Iteratee[Array[Byte], A]` but more precisely a `Iteratee[Array[Byte], Either[Result, A]]`, meaning that it has the opportunity to send directly an HTTP result itself (typically `400 BAD_REQUEST`, `412 PRECONDITION_FAILED` or `413 REQUEST_ENTITY_TOO_LARGE`) if it decides that it is not able to compute a correct value for the request body.

Once the body parser finishes its job and gives back a value of type `A`, the corresponding `Action` function is executed and the computed body value is passed into the request.

More about Actions

Previously we said that an `Action` was a `Request => Result` function. This is not entirely true. Let's have a more precise look at the `Action` trait:

```
trait Action[A] extends (Request[A] => Result) {  
    def parser: BodyParser[A]  
}
```

First we see that there is a generic type `A`, and then that an action must define a `BodyParser[A]`. With `Request[A]` being defined as:

```
trait Request[+A] extends RequestHeader {  
    def body: A  
}
```

The `A` type is the type of the request body. We can use any Scala type as the request body, for example `String`, `NodeSeq`, `Array[Byte]`, `JsonValue`, or `java.io.File`, as long as we have a body parser able to process it.

To summarize, an `Action[A]` uses a `BodyParser[A]` to retrieve a value of type `A` from the HTTP request, and to build a `Request[A]` object that is passed to the action code.

Default body parser: AnyContent

In our previous examples we never specified a body parser. So how can it work? If you don't specify your own body parser, Play will use the default, which processes the body as an instance of `play.api.mvc.AnyContent`.

This body parser checks the `Content-Type` header and decides what kind of body to process:

- `text/plain`: `String`
- `application/json`: `JsValue`
- `application/xml`, `text/xml` or `application/XXX+xml`: `NodeSeq`
- `application/form-url-encoded`: `Map[String, Seq[String]]`
- `multipart/form-data`: `MultipartFormData[TemporaryFile]`
- any other content type: `RawBuffer`

For example:

```

def save = Action { request =>
  val body: AnyContent = request.body
  val textBody: Option[String] = body.asText

  // Expecting text body
  textBody.map { text =>
    Ok("Got: " + text)
  }.getOrElse {
    BadRequest("Expecting text/plain request body")
  }
}

```

Specifying a body parser

The body parsers available in Play are defined in `play.api.mvc.BodyParsers.parse`.

So for example, to define an action expecting a text body (as in the previous example):

```

def save = Action(parse.text) { request =>
  Ok("Got: " + request.body)
}

```

Do you see how the code is simpler? This is because the `parse.text` body parser already sent a `400 BAD_REQUEST` response if something went wrong. We don't have to check again in our action code, and we can safely assume that `request.body` contains the valid `String` body.

Alternatively we can use:

```

def save = Action(parse.tolerantText) { request =>
  Ok("Got: " + request.body)
}

```

This one doesn't check the `Content-Type` header and always loads the request body as a `String`.

Tip: There is a `tolerant` fashion provided for all body parsers included in Play.

Here is another example, which will store the request body in a file:

```

def save = Action(parse.file(to = new File("/tmp/upload"))) { request =>
  Ok("Saved the request content to " + request.body)
}

```

Combining body parsers

In the previous example, all request bodies are stored in the same file. This is a bit problematic isn't it? Let's write another custom body parser that extracts the user name from the request Session, to give a unique file for each user:

```

val storeInUserFile = parse(using { request =>
  request.session.get("username").map { user =>
    file(to = new File("/tmp/" + user + ".upload"))
  }.getOrElse {
    sys.error("You don't have the right to upload here")
  }
}

def save = Action(storeInUserFile) { request =>
  Ok("Saved the request content to " + request.body)
}

```

Note: Here we are not really writing our own BodyParser, but just combining existing ones. This is often enough and should cover most use cases. Writing a `BodyParser` from scratch is covered in the advanced topics section.

Max content length

Text based body parsers (such as `text`, `json`, `xml` or `formUrlEncoded`) use a max content length because they have to load all the content into memory. By default, the maximum content length that they will parse is 100KB. It can be overridden by specifying the `play.http.parser.maxMemoryBuffer` property in `application.conf`:

`play.http.parser.maxMemoryBuffer=128K`

For parsers that buffer content on disk, such as the raw parser or `multipart/form-data`, the maximum content length is specified using

the `play.http.parser.maxDiskBuffer` property, it defaults to 10MB.

The `multipart/form-data` parser also enforces the text max length property for the aggregate of the data fields.

You can also override the default maximum length for a given action:

```

// Accept only 10KB of data.
def save = Action(parse.text(maxLength = 1024 * 10)) { request =>
  Ok("Got: " + text)
}

```

You can also wrap any body parser with `maxLength`:

```

// Accept only 10KB of data.
def save = Action(parse.maxLength(1024 * 10, storeInUserFile)) { request =>
  Ok("Saved the request content to " + request.body)
}

```

Next: Actions composition

Action composition

This chapter introduces several ways of defining generic action functionality.

Custom action builders

We saw [previously](#) that there are multiple ways to declare an action - with a request parameter, without a request parameter, with a body parser etc. In fact there are more than this, as we'll see in the chapter on [asynchronous programming](#).

These methods for building actions are actually all defined by a trait called `ActionBuilder` and the `Action` object that we use to declare our actions is just an instance of this trait. By implementing your own `ActionBuilder`, you can declare reusable action stacks, that can then be used to build actions.

Let's start with the simple example of a logging decorator, we want to log each call to this action.

The first way is to implement this functionality in the `invokeBlock` method, which is called for every action built by the `ActionBuilder`:

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

```
object LoggingAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    Logger.info("Calling action")
    block(request)
  }
}
```

Now we can use it the same way we use `Action`:

```
def index = LoggingAction {
  Ok("Hello World")
}
```

Since `ActionBuilder` provides all the different methods of building actions, this also works with, for example, declaring a custom body parser:

```
def submit = LoggingAction(parse.text) { request =>
  Ok("Got a body " + request.body.length + " bytes long")
}
```

Composing actions

In most applications, we will want to have multiple action builders, some that do different types of authentication, some that provide different types of generic functionality, etc. In which case, we won't want to rewrite our logging action code for each type of action builder, we will want to define it in a reuseable way.

Reusable action code can be implemented by wrapping actions:

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

```

case class Logging[A](action: Action[A]) extends Action[A] {

  def apply(request: Request[A]): Future[Result] = {
    Logger.info("Calling action")
    action(request)
  }

  lazy val parser = action.parser
}

```

We can also use the `Action` action builder to build actions without defining our own action class:

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

```

def logging[A](action: Action[A]) = Action.async(action.parser) { request =>
  Logger.info("Calling action")
  action(request)
}

```

Actions can be mixed in to action builders using the `composeAction` method:

```

object LoggingAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    block(request)
  }
  override def composeAction[A](action: Action[A]) = new Logging(action)
}

```

Now the builder can be used in the same way as before:

```

def index = LoggingAction {
  Ok("Hello World")
}

```

We can also mix in wrapping actions without the action builder:

```

def index = Logging {
  Action {
    Ok("Hello World")
  }
}

```

More complicated actions

So far we've only shown actions that don't impact the request at all. Of course, we can also read and modify the incoming request object:

```

import play.api.mvc._

def xForwardedFor[A](action: Action[A]) = Action.async(action.parser) { request =>
  val newRequest = request.headers.get("X-Forwarded-For").map { xff =>
    new WrappedRequest[A](request) {
      override def remoteAddress = xff
    }
  } getOrElse request
}

```

```
    action(newRequest)
}
```

Note: Play already has built in support for `X-Forwarded-For` headers.

We could block the request:

```
import play.api.mvc._

def onlyHttps[A](action: Action[A]) = Action.async(action.parser) { request =>
  request.headers.get("X-Forwarded-Proto").collect {
    case "https" => action(request)
  } getOrElse {
    Future.successful(FORBIDDEN("Only HTTPS requests allowed"))
  }
}
```

And finally we can also modify the returned result:

```
import play.api.mvc._
import play.api.libs.concurrent.Execution.Implicits._

def addUaHeader[A](action: Action[A]) = Action.async(action.parser) { request =>
  action(request).map(_.withHeaders("X-UA-Compatible" -> "Chrome=1"))
}
```

Different request types

While action composition allows you to perform additional processing at the HTTP request and response level, often you want to build pipelines of data transformations that add context to or perform validation on the request itself. `ActionFunction` can be thought of as a function on the request, parameterized over both the input request type and the output type passed on to the next layer. Each action function may represent modular processing such as authentication, database lookups for objects, permission checks, or other operations that you wish to compose and reuse across actions.

There are a few pre-defined traits implementing `ActionFunction` that are useful for different types of processing:

- `ActionTransformer` can change the request, for example by adding additional information.
- `ActionFilter` can selectively intercept requests, for example to produce errors, without changing the request value.
- `ActionRefiner` is the general case of both of the above.
- `ActionBuilder` is the special case of functions that take `Request` as input, and thus can build actions.

You can also define your own arbitrary `ActionFunction` by implementing the `invokeBlock` method. Often it is convenient to make the input and output types instances of `Request` (using `WrappedRequest`), but this is not strictly necessary.

Authentication

One of the most common use cases for action functions is authentication. We can easily implement our own authentication action transformer that determines the user from the original request and adds it to a new `UserRequest`. Note that this is also an `ActionBuilder` because it takes a simple `Request` as input:

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

```
class UserRequest[A](val username: Option[String], request: Request[A]) extends WrappedRequest[A](request)

object UserAction extends ActionBuilder[UserRequest] with ActionTransformer[Request, UserRequest] {
  def transform[A](request: Request[A]) = Future.successful {
    new UserRequest(request.session.get("username"), request)
  }
}
```

Play also provides a built in authentication action builder. Information on this and how to use it can be found [here](#).

Note: The built in authentication action builder is just a convenience helper to minimise the code necessary to implement authentication for simple cases, its implementation is very similar to the example above.

If you have more complex requirements than can be met by the built in authentication action, then implementing your own is not only simple, it is recommended.

Adding information to requests

Now let's consider a REST API that works with objects of type `Item`. There may be many routes under the `/item/:itemId` path, and each of these need to look up the item. In this case, it may be useful to put this logic into an action function.

First of all, we'll create a request object that adds an `Item` to our `UserRequest`:

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

```
class ItemRequest[A](val item: Item, request: UserRequest[A]) extends WrappedRequest[A](request) {
  def username = request.username
}
```

Now we'll create an action refiner that looks up that item and returns `Either` an error (`Left`) or a new `ItemRequest` (`Right`). Note that this action refiner is defined inside a method that takes the id of the item:

```
def ItemAction(itemId: String) = new ActionRefiner[UserRequest, ItemRequest] {
  def refine[A](input: UserRequest[A]) = Future.successful {
    ItemDao.findById(itemId)
      .map(new ItemRequest(_, input))
      .toRight(NotFound)
  }
}
```

Validating requests

Finally, we may want an action function that validates whether a request should continue. For example, perhaps we want to check whether the user from `UserAction` has permission to access the item from `ItemAction`, and if not return an error:

```
object PermissionCheckAction extends ActionFilter[ItemRequest] {
    def filter[A](input: ItemRequest[A]) = Future.successful {
        if (!input.item.accessibleByUser(input.username))
            Some(FORBIDDEN)
        else
            None
    }
}
```

Putting it all together

Now we can chain these action functions together (starting with an `ActionBuilder`) using `andThen` to create an action:

```
def tagItem(itemId: String, tag: String) =
    (UserAction andThen ItemAction(itemId) andThen PermissionCheckAction) { request =>
    request.item.addTag(tag)
    Ok("User " + request.username + " tagged " + request.item.id)
}
```

Play also provides a [global filter API](#), which is useful for global cross cutting concerns.

Next: [Content negotiation](#)

Content negotiation

Content negotiation is a mechanism that makes it possible to serve different representation of a same resource (URI). It is useful *e.g.* for writing Web Services supporting several output formats (XML, JSON, etc.). Server-driven negotiation is essentially performed using the `Accept*` requests headers. You can find more information on content negotiation in the [HTTP specification](#).

Language

You can get the list of acceptable languages for a request using the `play.api.mvc.RequestHeader#acceptLanguages` method that retrieves them from the `Accept-Language` header and sorts them according to their quality value. Play uses it in the `play.api.mvc.Controller#lang` method that provides an implicit `play.api.i18n.Lang` value to your actions, so they automatically use the best possible language (if supported by your application, otherwise your application's default language is used).

Content

Similarly, the `play.api.mvc.RequestHeader#acceptedTypes` method gives the list of acceptable result's MIME types for a request. It retrieves them from the `Accept` request header and sorts them according to their quality factor.

Actually, the `Accept` header does not really contain MIME types but media ranges (*e.g.* a request accepting all text results may set the `text/*` range, and the `*/*` range means that all result types are acceptable). Controllers provide a higher-level `render` method to help you to handle media ranges. Consider for example the following action definition:

```
val list = Action { implicit request =>
  val items = Item.findAll
  render {
    case Accepts.Html() => Ok(views.html.list(items))
    case Accepts.Json() => Ok(Json.toJson(items))
  }
}
```

`Accepts.Html()` and `Accepts.Json()` are extractors testing if a given media range matches `text/html` and `application/json`, respectively. The `render` method takes a partial function from `play.api.http.MediaRange` to `play.api.mvc.Result` and tries to apply it to each media range found in the request `Accept` header, in order of preference. If none of the acceptable media ranges is supported by your function, the `NotAcceptable` result is returned.

For example, if a client makes a request with the following value for the `Accept` header: `*/*;q=0.5, application/json`, meaning that it accepts any result type but prefers JSON, the above code will return the JSON representation. If another client makes a request with the following value for the `Accept` header: `application/xml`, meaning that it only accepts XML, the above code will return `NotAcceptable`.

Request extractors

See the API documentation of the `play.api.mvc.AcceptExtractors.Accepts` object for the list of the MIME types supported by Play out of the box in the `render` method. You can easily create your own extractor for a given MIME type using the `play.api.mvc.Accepting` case class, for example the following code creates an extractor checking that a media range matches the `audio/mp3` MIME type:

```
val AcceptsMp3 = Accepting("audio/mp3")
render {
  case AcceptsMp3() => ???
}
```

Next: [Handling errors](#)

Handling errors

There are two main types of errors that an HTTP application can return - client errors and server errors. Client errors indicate that the connecting client has done something wrong, server errors indicate that there is something wrong with the server.

Play will in many circumstances automatically detect client errors - these include errors such as malformed header values, unsupported content types, and requests for resources that can't be found. Play will also in many circumstances automatically handle server errors - if your action code throws an exception, Play will catch this and generate a server error page to send to the client.

The interface through which Play handles these errors is `HttpErrorHandler`. It defines two methods, `onClientError`, and `onServerError`.

Supplying a custom error handler

A custom error handler can be supplied by creating a class in the root package called `ErrorHandler` that implements `HttpErrorHandler`, for example:

```
import play.api.http.HttpErrorHandler
import play.api.mvc._
import play.api.mvc.Results._
import scala.concurrent._

class ErrorHandler extends HttpErrorHandler {

    def onClientError(request: RequestHeader, statusCode: Int, message: String) = {
        Future.successful(
            Status(statusCode)("A client error occurred: " + message)
        )
    }

    def onServerError(request: RequestHeader, exception: Throwable) = {
        Future.successful(
            InternalServerError("A server error occurred: " + exception.getMessage)
        )
    }
}
```

If you don't want to place your error handler in the root package, or if you want to be able to configure different error handlers for different environments, you can do this by configuring the `play.http.errorHandler` configuration property in `application.conf`:

```
play.http.errorHandler = "com.example.ErrorHandler"
```

Extending the default error handler

Out of the box, Play's default error handler provides a lot of useful functionality. For example, in dev mode, when a server error occurs, Play will attempt to locate and render the piece of code in your application that caused that exception, so that you can quickly see and identify the problem. You may want to provide custom server errors in production, while still maintaining that functionality in development. To facilitate this, Play provides a `DefaultHttpErrorHandler` that has some convenience methods that you can override so that you can mix in your custom logic with Play's existing behavior.

For example, to just provide a custom server error message in production, leaving the development error message untouched, and you also wanted to provide a specific forbidden error page:

```
import javax.inject._

import play.api.http.DefaultHttpErrorHandler
import play.api._
import play.api.mvc._
import play.api.mvc.Results._
import play.api.routing.Router
import scala.concurrent._

class ErrorHandler @Inject() (
    env: Environment,
    config: Configuration,
    sourceMapper: OptionalSourceMapper,
    router: Provider[Router]
) extends DefaultHttpErrorHandler(env, config, sourceMapper, router) {

    override def onProdServerError(request: RequestHeader, exception: UsefulException) = {
        Future.successful(
            InternalServerError("A server error occurred: " + exception.getMessage)
        )
    }

    override def onForbidden(request: RequestHeader, message: String) = {
        Future.successful(
            Forbidden("You're not allowed to access this resource.")
        )
    }
}
```

Checkout the full API documentation for `DefaultHttpErrorHandler` to see what methods are available to override, and how you can take advantage of them.

Next: [Asynchronous HTTP programming](#)

Handling asynchronous results

Make controllers asynchronous

Internally, Play Framework is asynchronous from the bottom up. Play handles every request in an asynchronous, non-blocking way.

The default configuration is tuned for asynchronous controllers. In other words, the application code should avoid blocking in controllers, i.e., having the controller code wait for an operation. Common examples of such blocking operations are JDBC calls, streaming API, HTTP requests and long computations.

Although it's possible to increase the number of threads in the default execution context to allow more concurrent requests to be processed by blocking controllers, following the recommended approach of keeping the controllers asynchronous makes it easier to scale and to keep the system responsive under load.

Creating non-blocking actions

Because of the way Play works, action code must be as fast as possible, i.e., non-blocking. So what should we return as result if we are not yet able to generate it? The response is a *future* result!

A `Future[Result]` will eventually be redeemed with a value of type `Result`. By giving a `Future[Result]` instead of a normal `Result`, we are able to quickly generate the result without blocking. Play will then serve the result as soon as the promise is redeemed.

The web client will be blocked while waiting for the response, but nothing will be blocked on the server, and server resources can be used to serve other clients.

How to create a `Future[Result]`

To create a `Future[Result]` we need another future first: the future that will give us the actual value we need to compute the result:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

```
val futurePIValue: Future[Double] = computePIAsynchronously()
val futureResult: Future[Result] = futurePIValue.map { pi =>
  Ok("PI value computed: " + pi)
}
```

All of Play's asynchronous API calls give you a `Future`. This is the case whether you are calling an external web service using the `play.api.libs.WS` API, or using Akka to schedule asynchronous tasks or to communicate with actors using `play.api.libs.Akka`. Here is a simple way to execute a block of code asynchronously and to get a `Future`:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

```
val futureInt: Future[Int] = scala.concurrent.Future {
  intensiveComputation()
}
```

Note: It's important to understand which thread code runs on with futures. In the two code blocks above, there is an import on Plays default execution context. This is an implicit parameter that gets passed to all methods on the future API that accept callbacks. The execution context will often be equivalent to a thread pool, though not necessarily.

You can't magically turn synchronous IO into asynchronous by wrapping it in a `Future`. If you can't change the application's architecture to avoid blocking operations, at some point that operation will have to be executed, and that thread is going to block. So in addition to enclosing the operation in a `Future`, it's necessary to configure it to run in a separate execution context that has been configured with enough threads to deal with the expected concurrency. See [Understanding Play thread pools](#) for more information.

It can also be helpful to use Actors for blocking operations. Actors provide a clean model for handling timeouts and failures, setting up blocking execution contexts, and managing any state that may be associated with the service. Also Actors provide patterns like `ScatterGatherFirstCompletedRouter` to address simultaneous cache and database requests and allow remote execution on a cluster of backend servers. But an Actor may be overkill depending on what you need.

Returning futures

While we were using the `Action.apply` builder method to build actions until now, to send an asynchronous result we need to use the `Action.async` builder method:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

```
def index = Action.async {
  val futureInt = scala.concurrent.Future { intensiveComputation() }
  futureInt.map(i => Ok("Got result: " + i))
}
```

Actions are asynchronous by default

Play `actions` are asynchronous by default. For instance, in the controller code below, the `{ Ok(...)` part of the code is not the method body of the controller. It is an anonymous function that is being passed to the `Action` object's `apply` method, which creates an object of type `Action`. Internally, the anonymous function that you wrote will be called and its result will be enclosed in a `Future`.

```
val echo = Action { request =>
  Ok("Got request [" + request + "]")
}
```

Note: Both `Action.apply` and `Action.async` create `Action` objects that are handled internally in the same way. There is a single kind of `Action`, which is asynchronous, and not two kinds (a synchronous one and an asynchronous one). The `.async` builder is just a facility to simplify creating actions based on APIs that return a `Future`, which makes it easier to write non-blocking code.

Handling time-outs

It is often useful to handle time-outs properly, to avoid having the web browser block and wait if something goes wrong. You can easily compose a promise with a promise timeout to handle these cases:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.duration._

def index = Action.async {
  val futureInt = scala.concurrent.Future { intensiveComputation() }
  val timeoutFuture = play.api.libs.concurrent.Promise.timeout("Oops", 1.second)
  Future.firstCompletedOf(Seq(futureInt, timeoutFuture)).map {
    case i: Int => Ok("Got result: " + i)
    case t: String => InternalServerError(t)
  }
}
```

Next: [Streaming HTTP responses](#)

Streaming HTTP responses

Standard responses and Content-Length header

Since HTTP 1.1, to keep a single connection open to serve several HTTP requests and responses, the server must send the appropriate `Content-Length` HTTP header along with the response.

By default, you are not specifying a `Content-Length` header when you send back a simple result, such as:

```
def index = Action {  
    Ok("Hello World")  
}
```

Of course, because the content you are sending is well-known, Play is able to compute the content size for you and to generate the appropriate header.

Note that for text-based content it is not as simple as it looks, since the `Content-Length` header must be computed according the character encoding used to translate characters to bytes.

Actually, we previously saw that the response body is specified using

```
a play.api.libs.iteratee.Enumerator:
```

```
def index = Action {  
    Result(  
        header = ResponseHeader(200),  
        body = Enumerator("Hello World")  
    )  
}
```

This means that to compute the `Content-Length` header properly, Play must consume the whole enumerator and load its content into memory.

Sending large amounts of data

If it's not a problem to load the whole content into memory for simple Enumerators, what about large data sets? Let's say we want to return a large file to the web client.

Let's first see how to create an `Enumerator[Array[Byte]]` enumerating the file content:

```
val file = new java.io.File("/tmp/fileToServe.pdf")  
val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)
```

Now it looks simple right? Let's just use this enumerator to specify the response body:

```
def index = Action {  
  
    val file = new java.io.File("/tmp/fileToServe.pdf")  
}
```

```
val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)
```

```
Result(  
  header = ResponseHeader(200),  
  body = fileContent  
)  
}
```

Actually we have a problem here. As we don't specify the `Content-Length` header, Play will have to compute it itself, and the only way to do this is to consume the whole enumerator content and load it into memory, and then compute the response size.

That's a problem for large files that we don't want to load completely into memory. So to avoid that, we just have to specify the `Content-Length` header ourself.

```
def index = Action {
```

```
  val file = new java.io.File("/tmp/fileToServe.pdf")  
  val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)
```

```
  Result(  
    header = ResponseHeader(200, Map(CONTENT_LENGTH -> file.length.toString)),  
    body = fileContent  
)  
}
```

This way Play will consume the body enumerator in a lazy way, copying each chunk of data to the HTTP response as soon as it is available.

Serving files

Of course, Play provides easy-to-use helpers for common task of serving a local file:

```
def index = Action {  
  Ok.sendFile(new java.io.File("/tmp/fileToServe.pdf"))  
}
```

This helper will also compute the `Content-Type` header from the file name, and add the `Content-Disposition` header to specify how the web browser should handle this response. The default is to ask the web browser to download this file by adding the header `Content-Disposition: attachment; filename=fileToServe.pdf` to the HTTP response.

You can also provide your own file name:

```
def index = Action {  
  Ok.sendFile(  
    content = new java.io.File("/tmp/fileToServe.pdf"),  
    fileName = _ => "termsOfService.pdf"  
)
```

```

}

If you want to serve this file inline:
def index = Action {
  Ok.sendFile(
    content = new java.io.File("/tmp/fileToServe.pdf"),
    inline = true
  )
}

```

Now you don't have to specify a file name since the web browser will not try to download it, but will just display the file content in the web browser window. This is useful for content types supported natively by the web browser, such as text, HTML or images.

Chunked responses

For now, it works well with streaming file content since we are able to compute the content length before streaming it. But what about dynamically computed content, with no content size available?

For this kind of response we have to use **Chunked transfer encoding**.

Chunked transfer encoding is a data transfer mechanism in version 1.1 of the Hypertext Transfer Protocol (HTTP) in which a web server serves content in a series of chunks. It uses the `Transfer-Encoding` HTTP response header instead of the `Content-Length` header, which the protocol would otherwise require. Because the `Content-Length` header is not used, the server does not need to know the length of the content before it starts transmitting a response to the client (usually a web browser). Web servers can begin transmitting responses with dynamically-generated content before knowing the total size of that content.

The size of each chunk is sent right before the chunk itself, so that a client can tell when it has finished receiving data for that chunk. Data transfer is terminated by a final chunk of length zero.

https://en.wikipedia.org/wiki/Chunked_transfer_encoding

The advantage is that we can serve the data **live**, meaning that we send chunks of data as soon as they are available. The drawback is that since the web browser doesn't know the content size, it is not able to display a proper download progress bar.

Let's say that we have a service somewhere that provides a dynamic `InputStream` computing some data. First we have to create an `Enumerator` for this stream:

```

val data = getDataStream
val dataContent: Enumerator[Array[Byte]] = Enumerator.fromStream(data)

```

We can now stream these data using a `Ok.chunked`:

```

def index = Action {
  val data = getDataStream
}

```

```
val dataContent: Enumerator[Array[Byte]] = Enumerator.fromStream(data)

Ok.chunked(dataContent)
}
```

Of course, we can use any `Enumerator` to specify the chunked data:

```
def index = Action {
  Ok.chunked(
    Enumerator("kiki", "foo", "bar").andThen(Enumerator.eof)
  )
}
```

We can inspect the HTTP response sent by the server:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

4
kiki
3
foo
3
bar
0
```

We get three chunks followed by one final empty chunk that closes the response.

[Next: Comet sockets](#)

Comet sockets

Using chunked responses to create Comet sockets

A good use for **Chunked responses** is to create Comet sockets. A Comet socket is just a chunked `text/html` response containing only `<script>` elements. At each chunk we write a `<script>` tag that is immediately executed by the web browser. This way we can send events live to the web browser from the server: for each message, wrap it into a `<script>` tag that calls a JavaScript callback function, and writes it to the chunked response.

Let's write a first proof-of-concept: an enumerator that generates `<script>` tags that each call the browser `console.log` JavaScript function:

```

def comet = Action {
  val events = Enumerator(
    """<script>console.log('kiki')</script>""",
    """<script>console.log('foo')</script>""",
    """<script>console.log('bar')</script>"""
  )
  Ok.chunked(events.as(HTML))
}

```

If you run this action from a web browser, you will see the three events logged in the browser console.

We can write this in a better way by using `play.api.libs.iteratee.Enumeratee` that is just an adapter to transform an `Enumerator[A]` into another `Enumerator[B]`. Let's use it to wrap standard messages into the `<script>` tags:

```

import play.twirl.api.Html

// Transform a String message into an Html script tag
val toCometMessage = Enumerator.map[String] { data =>
  Html("""<script>console.log("""" + data + """")</script>""")
}

```

```

def comet = Action {
  val events = Enumerator("kiki", "foo", "bar")
  Ok.chunked(events &> toCometMessage)
}

```

Tip: Writing `events &> toCometMessage` is just another way of writing `events.through(toCometMessage)`

Using the `play.api.libs.CometHelper`

We provide a Comet helper to handle these Comet chunked streams that do almost the same stuff that we just wrote.

Note: Actually it does more, like pushing an initial blank buffer data for browser compatibility, and it supports both String and JSON messages. It can also be extended via type classes to support more message types.

Let's just rewrite the previous example to use it:

```

def comet = Action {
  val events = Enumerator("kiki", "foo", "bar")
  Ok.chunked(events &> Comet(callback = "console.log"))
}

```

The forever iframe technique

The standard technique to write a Comet socket is to load an infinite chunked comet response in an HTML `iframe` and to specify a callback calling the parent frame:

```
def comet = Action {  
    val events = Enumerator("kiki", "foo", "bar")  
    Ok.chunked(events &> Comet(callback = "parent.cometMessage"))  
}
```

With an HTML page like:

```
<script type="text/javascript">  
var cometMessage = function(event) {  
    console.log(Received event: ' + event)  
}  
</script>  
  
<iframe src="/comet"></iframe>
```

Next: [WebSockets](#)

WebSockets

[WebSockets](#) are sockets that can be used from a web browser based on a protocol that allows two way full duplex communication. The client can send messages and the server can receive messages at any time, as long as there is an active WebSocket connection between the server and the client.

Modern HTML5 compliant web browsers natively support WebSockets via a JavaScript WebSocket API. However WebSockets are not limited in just being used by WebBrowsers, there are many WebSocket client libraries available, allowing for example servers to talk to each other, and also native mobile apps to use WebSockets. Using WebSockets in these contexts has the advantage of being able to reuse the existing TCP port that a Play server uses.

Handling WebSockets

Until now, we were using `Action` instances to handle standard HTTP requests and send back standard HTTP responses. WebSockets are a totally different beast and can't be handled via standard `Action`.

Play provides two different built in mechanisms for handling WebSockets. The first is using actors, the second is using iteratees. Both of these mechanisms can be accessed using the builders provided on [WebSocket](#).

Handling WebSockets with actors

To handle a WebSocket with an actor, we need to give Play a `akka.actor.Props` object that describes the actor that Play should create when it receives the WebSocket connection. Play will give us an `akka.actor.ActorRef` to send upstream messages to, so we can use that to help create the `Props` object:

```
import play.api.mvc._  
import play.api.Play.current  
  
def socket = WebSocket.acceptWithActor[String, String] { request => out =>  
    MyWebSocketActor.props(out)  
}
```

The actor that we're sending to here in this case looks like this:

```
import akka.actor._  
  
object MyWebSocketActor {  
    def props(out: ActorRef) = Props(new MyWebSocketActor(out))  
}  
  
class MyWebSocketActor(out: ActorRef) extends Actor {  
    def receive = {  
        case msg: String =>  
            out ! ("I received your message: " + msg)  
    }  
}
```

Any messages received from the client will be sent to the actor, and any messages sent to the actor supplied by Play will be sent to the client. The actor above simply sends every message received from the client back with `I received your message:` prepended to it.

Detecting when a WebSocket has closed

When the WebSocket has closed, Play will automatically stop the actor. This means you can handle this situation by implementing the actors `postStop` method, to clean up any resources the WebSocket might have consumed. For example:

```
override def postStop() = {  
    someResource.close()  
}
```

Closing a WebSocket

Play will automatically close the WebSocket when your actor that handles the WebSocket terminates. So, to close the WebSocket, send a `PoisonPill` to your own actor:

```
import akka.actor.PoisonPill
```

```
self ! PoisonPill
```

Rejecting a WebSocket

Sometimes you may wish to reject a WebSocket request, for example, if the user must be authenticated to connect to the WebSocket, or if the WebSocket is associated with some resource, whose id is passed in the path, but no resource with that id exists. Play provides `tryAcceptWithActor` to address this, allowing you to return either a result (such as `forbidden`, or `not found`), or the actor to handle the WebSocket with:

```
import scala.concurrent.Future
import play.api.mvc._
import play.api.Play.current
```

```
def socket = WebSocket.tryAcceptWithActor[String, String] { request =>
  Future.successful(request.session.get("user")) match {
    case None => Left(FORBIDDEN)
    case Some(_) => Right(MyWebSocketActor.props)
  }
}
```

Handling different types of messages

So far we have only seen handling `String` frames. Play also has built in handlers for `Array[Byte]` frames, and `JsValue` messages parsed from `String` frames. You can pass these as the type parameters to the WebSocket creation method, for example:

```
import play.api.mvc._
import play.api.libs.json._
import play.api.Play.current
```

```
def socket = WebSocket.acceptWithActor[JsValue, JsValue] { request => out =>
  MyWebSocketActor.props(out)
}
```

You may have noticed that there are two type parameters, this allows us to handle differently typed messages coming in to messages going out. This is typically not useful with the lower level frame types, but can be useful if you parse the messages into a higher level type.

For example, let's say we want to receive JSON messages, and we want to parse incoming messages as `InEvent` and format outgoing messages as `OutEvent`. The first thing we want to do is create JSON formats for out `InEvent` and `OutEvent` types:

```
import play.api.libs.json._
```

```
implicit val inEventFormat = Json.format[InEvent]
implicit val outEventFormat = Json.format[OutEvent]
```

Now we can create WebSocket `FrameFormatter`'s for these types:

```
import play.api.mvc.WebSocket.FrameFormatter
```

```
implicit val inEventFrameFormatter = FrameFormatter.jsonFrame[InEvent]
implicit val outEventFrameFormatter = FrameFormatter.jsonFrame[OutEvent]
```

And finally, we can use these in our WebSocket:

```
import play.api.mvc._
import play.api.Play.current

def socket = WebSocket.acceptWithActor[InEvent, OutEvent] { request => out =>
  MyWebSocketActor.props(out)
}
```

Now in our actor, we will receive messages of type `InEvent`, and we can send messages of type `OutEvent`.

Handling WebSockets with iteratees

While actors are a better abstraction for handling discrete messages, iteratees are often a better abstraction for handling streams.

To handle a WebSocket request, use a `WebSocket` instead of an `Action`:

```
import play.api.mvc._
import play.api.libs.iteratee._
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

```
def socket = WebSocket.using[String] { request =>

  // Log events to the console
  val in = Iteratee.foreach[String](println).map { _ =>
    println("Disconnected")
  }

  // Send a single 'Hello!' message
  val out = Enumerator("Hello!")

  (in, out)
}
```

A `WebSocket` has access to the request headers (from the HTTP request that initiates the WebSocket connection), allowing you to retrieve standard headers and session data. However, it doesn't have access to a request body, nor to the HTTP response. When constructing a `WebSocket` this way, we must return both `in` and `out` channels.

- The `in` channel is an `Iteratee[A, Unit]` (where `A` is the message type - here we are using `String`) that will be notified for each message, and will receive `EOF` when the socket is closed on the client side.
- The `out` channel is an `Enumerator[A]` that will generate the messages to be sent to the Web client. It can close the connection on the server side by sending `EOF`.

In this example we are creating a simple iteratee that prints each message to console. To send messages, we create a simple dummy enumerator that will send a single `Hello!` message.

Tip: You can test WebSockets on <https://www.websocket.org/echo.html>. Just set the location to `ws://localhost:9000`.

Let's write another example that discards the input data and closes the socket just after sending the `Hello!` message:

```
import play.api.mvc._  
import play.api.libs.iteratee._
```

```
def socket = WebSocket.using[String] { request =>  
  
    // Just ignore the input  
    val in = Iteratee.ignore[String]  
  
    // Send a single 'Hello!' message and close  
    val out = Enumerator("Hello!").andThen(Enumerator.eof)  
  
    (in, out)  
}
```

Here is another example in which the input data is logged to standard out and broadcast to the client utilizing `Concurrent.broadcast`.

```
import play.api.mvc._  
import play.api.libs.iteratee._  
import play.api.libs.concurrent.Execution.Implicits.defaultContext  
  
def socket = WebSocket.using[String] { request =>  
  
    // Concurrent.broadcast returns (Enumerator, Concurrent.Channel)  
    val (out, channel) = Concurrent.broadcast[String]  
  
    // log the message to stdout and send response back to client  
    val in = Iteratee.foreach[String] {  
        msg =>  
        println(msg)  
        // the Enumerator returned by Concurrent.broadcast subscribes to the channel and will  
        // receive the pushed messages  
        channel push("I received your message: " + msg)  
    }  
    (in, out)  
}
```

Next: The template engine

The template engine

A type safe template engine based on Scala

Play comes with [Twirl](#), a powerful Scala-based template engine, whose design was inspired by ASP.NET Razor. Specifically it is:

- **compact, expressive, and fluid:** it 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:** it allows you to quickly become productive, with a minimum of concepts. You use simple Scala constructs and all your existing HTML skills.
- **not a new language:** we consciously chose not to create a new language. Instead we wanted to enable Scala developers to use their existing Scala language skills, and deliver a template markup syntax that enables an awesome HTML construction workflow.
- **editable in any text editor:** it doesn't require a specific tool and enables you to be productive in any plain old text editor.

Templates are compiled, so you will see any errors in your browser:

A screenshot of a web browser window titled "Compilation error". The address bar shows "localhost:9000". The main content area has a red header with the text "Compilation error". Below the header, there is a message: "value name is not a member of models.Customer". Underneath this message, it says "In /Volumes/Data/gbo/myFirstApp/app/views/index.scala.html at line 3." The code editor shows the following Scala code:

```
1 @(customer: Customer, orders: Seq[Order])
2
3 <h1>Welcome @customer.name!</h1>
4
5 <ul>
6 @orders.map { order =>
7   <li>@order.title</li>
```

Overview

A Play Scala template is a simple text file that contains small blocks of Scala code. Templates can generate any text-based format, such as HTML, XML or CSV.

The template system has been designed to feel comfortable to those used to working with HTML, allowing front-end developers to easily work with the templates.

Templates 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.html.Application.index` class that has an `apply()` method.

For example, here is a simple template:

```
@(customer: Customer, orders: List[Order])  
  
<h1>Welcome @customer.name!</h1>  
  
<ul>  
@for(order <- orders) {  
 <li>@order.title</li>  
}  
</ul>
```

You can then call this from any Scala code as you would normally call a method on a class:

```
val content = views.html.Application.index(c, o)
```

Syntax: the magic '@' character

The Scala template uses `@` as the single special character. Every time this character is encountered, it indicates the beginning of a dynamic statement. You are not required to explicitly close the code block - the end of the dynamic statement will be inferred from your code:

```
Hello @customer.name!  
~~~~~  
Dynamic code
```

Because the template engine automatically detects the end of your code block by analysing your code, this syntax only supports simple statements. If you want to insert a multi-token statement, explicitly mark it using brackets:

```
Hello @(customer.firstName + customer.lastName)!  
~~~~~  
Dynamic Code
```

You can also use curly brackets, to write a multi-statement block:

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

Because `@` is a special character, you'll sometimes need to escape it. Do this by using `@@`:
My email is bob@@example.com

Template parameters

A template is like a function, so it needs parameters, which must be declared at the top of the template file:

```
@(customer: Customer, orders: List[Order])
```

You can also use default values for parameters:

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

Or even several parameter groups:

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

Iterating

You can use the `for` keyword, in a pretty standard way:

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

Note: Make sure that `{` is on the same line with `for` to indicate that the expression continues to next line.

If-blocks

If-blocks are nothing special. Simply use Scala's standard `if` statement:

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

Declaring reusable blocks

You can create reusable code blocks:

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

<ul>
@for(product <- products) {
  @display(product)
}
</ul>
```

Note that you can also declare reusable pure code blocks:

```
@title(text: String) = @{
```

```
text.split(' ').map(_.capitalize).mkString(" ")
}
```

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

Note: Declaring code block this way in a template can be sometime useful but keep in mind that a template is not the best place to write complex logic. It is often better to externalize these kind of code in a Scala class (that you can store under the `views`/package as well if you want).

By convention a reusable block defined with a name starting with `implicit` will be marked as `implicit`:

```
@implicitFieldConstructor = @{ MyFieldConstructor() }
```

Declaring reusable values

You can define scoped values using the `defining` helper:

```
@defining(user.firstName + " " + user.lastName) { fullName =>
  <div>Hello @fullName</div>
}
```

Import statements

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

```
@(customer: Customer, orders: List[Order])
@import utils._
```

```
...
```

To make an absolute resolution, use `root` prefix in the import statement.

```
@import _root_.company.product.core._
```

If you have common imports, which you need in all templates, you can declare in `build.sbt`

```
TwirlKeys.templateImports += "org.abc.backend._"
```

Comments

You can write server side block comments in templates using `@* * @`:

```
@*****
* This is a comment *
*****@
```

You can put a comment on the first line to document your template into the Scala API doc:

```
@*****
* Home page.          *
*                      *
* @param msg The message to display *
```

```
*****@  
@(msg: String)  
  
<h1>@msg</h1>
```

Escaping

By default, dynamic content parts are escaped according to the template type's (e.g. HTML or XML) rules. If you want to output a raw content fragment, wrap it in the template content type.

For example to output raw HTML:

```
<p>  
 @Html(article.content)  
</p>
```

String interpolation

The template engine can be used as a [string interpolator](#). You basically trade the “@” for a “\$”:

```
import play.twirl.api.StringInterpolation  
  
val name = "Martin"  
val p = html"<p>Hello $name</p>"
```

[Next: Common use cases](#)

Scala templates common use cases

Templates, being simple functions, can be composed in any way you want. Below are examples of some common scenarios.

Layout

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

```
@(title: String)(content: Html)  
<!DOCTYPE html>
```

```

<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
  </body>
</html>

```

As you can see, this template takes two parameters: a title and an HTML content block.
Now we can use it from another `views/Application/index.scala.html` template:

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

```
  <h1>Home page</h1>
```

```
}
```

Note: We sometimes use named parameters (like `@main(title = "Home")`), sometimes not like `@main("Home")`. It is as you want, choose whatever is clearer in a specific context.

Sometimes you need a second page-specific content block for a sidebar or breadcrumb trail, for example. You can do this with an additional parameter:

```

@(title: String)(sidebar: Html)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="sidebar">@sidebar</section>
    <section class="content">@content</section>
  </body>
</html>

```

Using this from our ‘index’ template, we have:

```

@main("Home") {
  <h1>Sidebar</h1>

} {
  <h1>Home page</h1>

}

```

Alternatively, we can declare the sidebar block separately:

```

@sidebar = {
  <h1>Sidebar</h1>
}

@main("Home")(sidebar) {
  <h1>Home page</h1>
}

```

```
}
```

Tags (they are just functions, right?)

Let's write a simple `views/tags/notice.scala.html` tag that displays 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 now let's use it from another template:

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

Includes

Again, there's nothing special here. You can just call any other template you like (and in fact any other function coming from anywhere at all):

```
<h1>Home</h1>
```

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

moreScripts and moreStyles equivalents

To define old moreScripts or moreStyles variables equivalents (like on Play! 1.x) on a Scala template, you can define a variable in the main template like this :

```
@(title: String, scripts: Html = Html(""))(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>
    @scripts
  </head>
  <body>
    <div class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <a class="brand" href="#">Movies</a>
        </div>
      </div>
    <div class="container">
      @content
    </div>
  </body>
</html>
```

And on an extended template that need an extra script :

```
@scripts = {
  <script type="text/javascript">alert("hello !");</script>
}

@main("Title",scripts){

  Html content here ...

}
```

And on an extended template that not need an extra script, just like this :

```
@main("Title"){

    Html content here ...

}

Next: Custom format
```

Adding support for a custom format to the template engine

The built-in template engine supports common template formats (HTML, XML, etc.) but you can easily add support for your own formats, if needed. This page summarizes the steps to follow to support a custom format.

Overview of the templating process

The template engine builds its result by appending static and dynamic content parts of a template. Consider for instance the following template:

```
foo @bar baz
```

It consists in two static parts (`foo` and `baz`) around one dynamic part (`bar`). The template engine concatenates these parts together to build its result. Actually, in order to prevent cross-site scripting attacks, the value of `bar` can be escaped before being concatenated to the rest of the result. This escaping process is specific to each format: e.g. in the case of HTML you want to transform "<" into "<".

How does the template engine know which format correspond to a template file? It looks at its extension: e.g. if it ends with `.scala.html` it associates the HTML format to the file.

Finally, you usually want your template files to be used as the body of your HTTP responses, so you have to define how to make a Play result from a template rendering result.

In summary, to support your own template format you need to perform the following steps:

- Implement the text integration process for the format ;
- Associate a file extension to the format ;
- Eventually tell Play how to send the result of a template rendering as an HTTP response body.

Implement a format

Implement the `play.twirl.api.Format[A]` trait that has the methods `raw(text: String): A` and `escape(text: String): A` that will be used to integrate static and dynamic template parts, respectively.

The type parameter `A` of the format defines the result type of the template rendering, e.g. `Html` for a HTML template. This type must be a subtype of the `play.twirl.api.Appendable[A]` trait that defines how to concatenates parts together.

For convenience, Play provides a `play.twirl.api.BufferedContent[A]` abstract class that implements `play.twirl.api.Appendable[A]` using a `StringBuilder` to build its result and that implements the `play.twirl.api.Content` trait so Play knows how to serialize it as an HTTP response body (see the last section of this page for details). In short, you need to write to classes: one defining the result (implementing `play.twirl.api.Appendable[A]`) and one defining the text integration process (implementing `play.twirl.api.Format[A]`). For instance, here is how the HTML format is defined:

```
// The `Html` result type. We extend `BufferedContent[Html]` rather than just `Appendable[Html]` so
// Play knows how to make an HTTP result from a `Html` value
class Html(buffer: StringBuilder) extends BufferedContent[Html](buffer) {
    val contentType = MimeType.HTML
}

object HtmlFormat extends Format[Html] {
    def raw(text: String): Html = ...
    def escape(text: String): Html = ...
}
```

Associate a file extension to the format

The templates are compiled into a `.scala` files by the build process just before compiling the whole application sources. The `TwirlKeys.templateFormats` key is a sbt setting of type `Map[String, String]` defining the mapping between file extensions and template formats. For instance, if HTML was not supported out of the box by Play, you would have to write the following in your build file to associate the `.scala.html` files to the `play.twirl.api.HtmlFormat` format:

```
TwirlKeys.templateFormats += ("html" -> "my.HtmlFormat.instance")
```

Note that the right side of the arrow contains the fully qualified name of a value of type `play.twirl.api.Format[_]`.

Tell Play how to make an HTTP result from a template result type

Play can write an HTTP response body for any value of type `A` for which it exists an implicit `play.api.http.Writeable[A]` value. So all you need is to define such a value for your template result type. For instance, here is how to define such a value for HTTP:

```
implicit def writableHttp(implicit codec: Codec): Writeable[Http] =
```

```
  Writeable[Http](result => codec.encode(result.body), Some(ContentTypes.HTTP))
```

Note: if your template result type extends `play.twirl.api.BufferedContent` you only need to define an

```
implicit play.api.http.ContentTypeOf value:
```

```
scala implicit def contentTypeHttp(implicit codec: Codec):
```

```
ContentTypeOf[Http] = ContentTypeOf[Http](Some(ContentTypes.HTTP))
```

Next: [Form submission and validation](#)

Handling form submission

Overview

Form handling and submission is an important part of any web application. Play comes with features that make handling simple forms easy and complex forms possible.

Play's form handling approach is based around the concept of binding data. When data comes in from a POST request, Play will look for formatted values and bind them to a `Form` object. From there, Play can use the bound form to value a case class with data, call custom validations, and so on.

Typically forms are used directly from a `Controller` instance. However, `Form` definitions do not have to match up exactly with case classes or models: they are purely for handling input and it is reasonable to use a distinct `Form` for a distinct POST.

Imports

To use forms, import the following packages into your class:

```
import play.api.data._  
import play.api.data.Forms._
```

Form Basics

We'll go through the basics of form handling:

- defining a form,
- defining constraints in the form,
- validating the form in an action,
- displaying the form in a view template,
- and finally, processing the result (or errors) of the form in a view template.

The end result will look something like this:

user.scala.html

```
@(userForm:Form[UserData])  
  
@import helpers._  
  
@helper.form(action = routes.Application.userPost) {  
    @inputText(userForm("name"))  
    @inputText(userForm("age"))  
    <input type="submit" value="Submit">  
}
```

POST /user
name=bob
age=25

userForm:Form[UserData]

name = text
age = number

userForm.bindFromRequest

myData:UserData

name = bob
age = 25

userForm.fold(errorFunc, su

home.scala.html

```
@(user:User)  
  
Name: @user.name  
Age: @user.age
```

302 Redirect

user.name = myData.name
user.age = myData.age
User.store(user)

Defining a form

First, define a case class which contains the elements you want in the form. Here we want to capture the name and age of a user, so we create a `UserData` object:

```
case class UserData(name: String, age: Int)
```

Now that we have a case class, the next step is to define a `Form` structure. The function of a `Form` is to transform form data into a bound instance of a case class, and we define it like follows:

```
val userForm = Form(  
  mapping(  
    "name" -> text,  
    "age" -> number  
  )(UserData.apply)(UserData.unapply)  
)
```

The `Forms` object defines the `mapping` method. This method takes the names and constraints of the form, and also takes two functions: an `apply` function and an `unapply` function. Because `UserData` is a case class, we can plug its `apply` and `unapply` methods directly into the mapping method.

Note: Maximum number of fields for a single tuple or mapping is 22 due to the way form handling is implemented. If you have more than 22 fields in your form, you should break down your forms using lists or nested values.

A form will create `UserData` instance with the bound values when given a Map:

```
val anyData = Map("name" -> "bob", "age" -> "21")  
val userData = userForm.bind(anyData).get
```

But most of the time you'll use forms from within an Action, with data provided from the request. `Form` contains `bindFromRequest`, which will take a request as an implicit parameter. If you define an implicit request, then `bindFromRequest` will find it.

```
val userData = userForm.bindFromRequest.get
```

Note: There is a catch to using `get` here. If the form cannot bind to the data, then `get` will throw an exception. We'll show a safer way of dealing with input in the next few sections.

You are not limited to using case classes in your form mapping. As long as the apply and unapply methods are properly mapped, you can pass in anything you like, such as tuples using the `Forms.tuple` mapping or model case classes. However, there are several advantages to defining a case class specifically for a form:

- **Form specific case classes are convenient.** Case classes are designed to be simple containers of data, and provide out of the box features that are a natural match with `Form` functionality.
- **Form specific case classes are powerful.** Tuples are convenient to use, but do not allow for custom apply or unapply methods, and can only reference contained data by arity (`1`, `2`, etc.)
- **Form specific case classes are targeted specifically to the Form.** Reusing model case classes can be convenient, but often models will contain additional domain logic and even persistence details that can lead to tight coupling. In addition, if there is not a direct 1:1 mapping between the form and the model, then sensitive fields must be explicitly ignored to prevent a `parameter tampering` attack.

Defining constraints on the form

The `text` constraint considers empty strings to be valid. This means that `name` could be empty here without an error, which is not what we want. A way to ensure that `name` has the appropriate value is to use the `nonEmptyText` constraint.

```
val userFormConstraints2 = Form(  
  mapping(  
    "name" -> nonEmptyText,  
    "age" -> number(min = 0, max = 100)
```

```
)(UserData.apply)(UserData.unapply)  
)
```

Using this form will result in a form with errors if the input to the form does not match the constraints:

```
val boundForm = userFormConstraints2.bind(Map("bob" -> "", "age" -> "25"))  
boundForm.hasErrors must beTrue
```

The out of the box constraints are defined on the **Forms** object:

- **text**: maps to `scala.String`, optionally takes `minLength` and `maxLength`.
- **nonEmptyText**: maps to `scala.String`, optionally takes `minLength` and `maxLength`.
- **number**: maps to `scala.Int`, optionally takes `min`, `max`, and `strict`.
- **longNumber**: maps to `scala.Long`, optionally takes `min`, `max`, and `strict`.
- **bigDecimal**: takes `precision` and `scale`.
- **date**, **sqlDate**, **jodaDate**: maps to `java.util.Date`, `java.sql.Date` and `org.joda.time.DateTime`, optionally takes `pattern` and `timeZone`.
- **jodaLocalDate**: maps to `org.joda.time.LocalDate`, optionally takes `pattern`.
- **email**: maps to `scala.String`, using an email regular expression.
- **boolean**: maps to `scala.Boolean`.
- **checked**: maps to `scala.Boolean`.
- **optional**: maps to `scala.Option`.

Defining ad-hoc constraints

You can define your own ad-hoc constraints on the case classes using the **validation package**.

```
val userFormConstraints = Form(  
  mapping(  
    "name" -> text.verifying(nonEmpty),  
    "age" -> number.verifying(min(0), max(100))  
  )(UserData.apply)(UserData.unapply)  
)
```

You can also define ad-hoc constraints on the case classes themselves:

```
def validate(name: String, age: Int) = {  
  name match {  
    case "bob" if age >= 18 =>  
      Some(UserData(name, age))  
    case "admin" =>  
      Some(UserData(name, age))  
    case _ =>  
      None  
  }  
}  
  
val userFormConstraintsAdHoc = Form(  
  mapping(  
    "name" -> text,
```

```

"age" -> number
)(UserData.apply)(UserData.unapply) verifying("Failed form constraints!", fields => fields match {
  case userData => validate(userData.name, userData.age).isDefined
})
)
)

```

You also have the option of constructing your own custom validations. Please see the [custom validations](#) section for more details.

Validating a form in an Action

Now that we have constraints, we can validate the form inside an action, and process the form with errors.

We do this using the `fold` method, which takes two functions: the first is called if the binding fails, and the second is called if the binding succeeds.

```

userForm.bindFromRequest.fold(
  formWithErrors => {
    // binding failure, you retrieve the form containing errors:
    BadRequest(views.html.user(formWithErrors))
  },
  userData => {
    /* binding success, you get the actual value. */
    val newUser = models.User(userData.name, userData.age)
    val id = models.User.create(newUser)
    Redirect(routes.Application.home(id))
  }
)

```

In the failure case, we render the page with `BadRequest`, and pass in the form *with errors* as a parameter to the page. If we use the view helpers (discussed below), then any errors that are bound to a field will be rendered in the page next to the field.

In the success case, we're sending a `Redirect` with a route

`to routes.Application.home` here instead of rendering a view template. This pattern is called [Redirect after POST](#), and is an excellent way to prevent duplicate form submissions.

Note: “Redirect after POST” is **required** when using `flashing` or other methods with `flash scope`, as new cookies will only be available after the redirected HTTP request.

Alternatively, you can use the `parse.form` body parser that binds the content of the request to your form.

```

val userPost = Action(parse.form(userForm)) { implicit request =>
  val userData = request.body
  val newUser = models.User(userData.name, userData.age)
  val id = models.User.create(newUser)
  Redirect(routes.Application.home(id))
}

```

In the failure case, the default behaviour is to return an empty `BadRequest` response. You can override this behaviour with your own logic. For instance, the following code is completely equivalent to the preceding one using `bindFromRequest` and `fold`.

```

val userPostWithErrors = Action(parse.form(userForm, onErrors = (formWithErrors: Form[UserData]) =>
  BadRequest(views.html.user(formWithErrors))))
{ implicit request =>
}

```

```

    val userData = request.body
    val newUser = models.User(userData.name, userData.age)
    val id = models.User.create(newUser)
    Redirect(routes.Application.home(id))
}

```

Showing forms in a view template

Once you have a form, then you need to make it available to the template engine. You do this by including the form as a parameter to the view template. For `user.scala.html`, the header at the top of the page will look like this:

```
@(userForm: Form[UserData])(implicit messages: Messages)
```

Because `user.scala.html` needs a form passed in, you should pass the

`emptyUserForm` initially when rendering `user.scala.html`:

```

def index = Action {
  Ok(views.html.user(userForm))
}

```

The first thing is to be able to create the `form tag`. It is a simple view helper that creates a `form tag` and sets the `action` and `method` tag parameters according to the reverse route you pass in:

```

@helper.form(action = routes.Application.userPost()) {
  @helper.inputText(userForm("name"))
  @helper.inputText(userForm("age"))
}

```

You can find several input helpers in the `views.html.helper` package. You feed them with a form field, and they display the corresponding HTML input, setting the value, constraints and displaying errors when a form binding fails.

Note: You can use `@import helper._` in the template to avoid prefixing helpers with `@helper`.

There are several input helpers, but the most helpful are:

- `form`: renders a `form` element.
- `inputText`: renders a `text input` element.
- `inputPassword`: renders a `password input` element.
- `inputDate`: renders a `date input` element.
- `inputFile`: renders a `file input` element.
- `inputRadioGroup`: renders a `radio input` element.
- `select`: renders a `select` element.
- `textarea`: renders a `textarea` element.
- `checkbox`: renders a `checkbox` element.
- `input`: renders a generic input element (which requires explicit arguments).

As with the `form` helper, you can specify an extra set of parameters that will be added to the generated Html:

```
@helper.inputText(userForm("name"), 'id -> "name", 'size -> 30)
```

The generic `input` helper mentioned above will let you code the desired HTML result:

```
@helper.input(userForm("name")) { (id, name, value, args) =>
```

```

<input type="text" name="@name" id="@id" @toHtmlArgs(args)>
}

```

Note: All extra parameters will be added to the generated Html, unless they start with the _ character.
Arguments starting with _ are reserved for **field constructor arguments**.

For complex form elements, you can also create your own custom view helpers (using scala classes in the `views` package) and **custom field constructors**.

Displaying errors in a view template

The errors in a form take the form of `Map[String, FormError]` where `FormError` has:

- `key`: should be the same as the field.
- `message`: a message or a message key.
- `args`: a list of arguments to the message.

The form errors are accessed on the bound form instance as follows:

- `errors`: returns all errors as `Seq[FormError]`.
- `globalErrors`: returns errors without a key as `Seq[FormError]`.
- `error("name")`: returns the first error bound to key as `Option[FormError]`.
- `errors("name")`: returns all errors bound to key as `Seq[FormError]`.

Errors attached to a field will render automatically using the form helpers,
so `@helper.inputText` with errors can display as follows:

```

<dl class="error" id="age_field">
  <dt><label for="age">Age:</label></dt>
  <dd><input type="text" name="age" id="age" value=""></dd>
  <dd class="error">This field is required!</dd>
  <dd class="error">Another error</dd>
  <dd class="info">Required</dd>
  <dd class="info">Another constraint</dd>
</dl>

```

Global errors that are not bound to a key do not have a helper and must be defined explicitly in the page:

```

@if(userForm.hasGlobalErrors) {
  <ul>
    @for(error <- userForm.globalErrors) {
      <li>@error.message</li>
    }
  </ul>
}

```

Mapping with tuples

You can use tuples instead of case classes in your fields:

```

val userFormTuple = Form(
  tuple(
    "name" -> text,
    "age" -> number
  )
)

```

```
) // tuples come with built-in apply/unapply  
)
```

Using a tuple can be more convenient than defining a case class, especially for low arity tuples:

```
val anyData = Map("name" -> "bob", "age" -> "25")  
val (name, age) = userFormTuple.bind(anyData).get
```

Mapping with single

Tuples are only possible when there are multiple values. If there is only one field in the form, use `Forms.single` to map to a single value without the overhead of a case class or tuple:

```
val singleForm = Form(  
  single(  
    "email" -> email  
  )  
)
```

```
val emailValue = singleForm.bind(Map("email" -> "bob@example.com")).get
```

Fill values

Sometimes you'll want to populate a form with existing values, typically for editing data:

```
val filledForm = userForm.fill(UserData("Bob", 18))
```

When you use this with a view helper, the value of the element will be filled with the value:

```
@helper.inputText(filledForm("name")) @* will render value="Bob" *@
```

Fill is especially helpful for helpers that need lists or maps of values, such as the `select` and `inputRadioGroup` helpers. Use `options` to value these helpers with lists, maps and pairs.

Nested values

A form mapping can define nested values by using `Forms.mapping` inside an existing mapping:

```
case class AddressData(street: String, city: String)  
  
case class UserAddressData(name: String, address: AddressData)  
val userFormNested: Form[UserAddressData] = Form(  
  mapping(  
    "name" -> text,  
    "address" -> mapping(  
      "street" -> text,  
      "city" -> text  
    )(AddressData.apply)(AddressData.unapply)  
  )(UserAddressData.apply)(UserAddressData.unapply)  
)
```

Note: When you are using nested data this way, the form values sent by the browser must be named like `address.street`, `address.city`, etc.

```
@helper.inputText(userFormNested("name"))
@helper.inputText(userFormNested("address.street"))
@helper.inputText(userFormNested("address.city"))
```

Repeated values

A form mapping can define repeated values using `Forms.list` or `Forms.seq`:

```
case class UserListData(name: String, emails: List[String])
val userFormRepeated = Form(
  mapping(
    "name" -> text,
    "emails" -> list(email)
  )(UserListData.apply)(UserListData.unapply)
)
```

When you are using repeated data like this, there are two alternatives for sending the form values in the HTTP request. First, you can suffix the parameter with an empty bracket pair, as in “`emails[]`”. This parameter can then be repeated in the standard way, as in `http://foo.com/request?emails[] = a@b.com&emails[] = c@d.com`.

Alternatively, the client can explicitly name the parameters uniquely with array subscripts, as in `emails[0]`, `emails[1]`, `emails[2]`, and so on. This approach also allows you to maintain the order of a sequence of inputs.

If you are using Play to generate your form HTML, you can generate as many inputs for the `emails` field as the form contains, using the `repeat` helper:

```
@helper.inputText(myForm("name"))
@helper.repeat(myForm("emails"), min = 1) { emailField =>
  @helper.inputText(emailField)
}
```

The `min` parameter allows you to display a minimum number of fields even if the corresponding form data are empty.

Optional values

A form mapping can also define optional values using `Forms.optional`:

```
case class UserOptionalData(name: String, email: Option[String])
val userFormOptional = Form(
  mapping(
    "name" -> text,
    "email" -> optional(email)
  )(UserOptionalData.apply)(UserOptionalData.unapply)
)
```

This maps to an `Option[A]` in output, which is `None` if no form value is found.

Default values

You can populate a form with initial values using `Form#fill`:

```
val filledForm = userForm.fill(UserData("Bob", 18))
```

Or you can define a default mapping on the number using `Forms.default`:

```
Form(
  mapping(
    "name" -> default(text, "Bob")
    "age" -> default(number, 18)
  )(User.apply)(User.unapply)
)
```

Ignored values

If you want a form to have a static value for a field, use `Forms.ignored`:

```
val userFormStatic = Form(  
  mapping(  
    "id" -> ignored(23L),  
    "name" -> text,  
    "email" -> optional(email)  
  )(UserStaticData.apply)(UserStaticData.unapply)  
)
```

Putting it all together

Here's an example of what a model and controller would look like for managing an entity.

Given the case class `Contact`:

```
case class Contact(firstname: String,  
                   lastname: String,  
                   company: Option[String],  
                   informations: Seq[ContactInformation])
```

```
object Contact {  
  def save(contact: Contact): Int = 99  
}
```

```
case class ContactInformation(label: String,  
                             email: Option[String],  
                             phones: List[String])
```

Note that `Contact` contains a `Seq` with `ContactInformation` elements and a `List` of `String`. In this case, we can combine the nested mapping with repeated mappings (defined with `Forms.seq` and `Forms.list`, respectively).

```
val contactForm: Form[Contact] = Form(  
  
  // Defines a mapping that will handle Contact values  
  mapping(  
    "firstname" -> nonEmptyText,  
    "lastname" -> nonEmptyText,  
    "company" -> optional(text),  
  
    // Defines a repeated mapping  
    "informations" -> seq(  
      mapping(  
        "label" -> nonEmptyText,  
        "email" -> optional(email),  
        "phones" -> list(  
          text verifying pattern("""[0-9.+]+""".r, error="A valid phone number is required")  
        )  
      )(ContactInformation.apply)(ContactInformation.unapply)  
    )  
  )(Contact.apply)(Contact.unapply)
```

)

And this code shows how an existing contact is displayed in the form using filled data:

```
def editContact = Action {  
    val existingContact = Contact(  
        "Fake", "Contact", Some("Fake company"), informations = List(  
            ContactInformation(  
                "Personal", Some("fakecontact@gmail.com"), List("01.23.45.67.89", "98.76.54.32.10")  
,  
            ContactInformation(  
                "Professional", Some("fakecontact@company.com"), List("01.23.45.67.89")  
,  
            ContactInformation(  
                "Previous", Some("fakecontact@oldcompany.com"), List()  
)  
)  
)  
)  
Ok(views.html.contact.form(contactForm.fill(existingContact)))  
}
```

Finally, this is what a form submission handler would look like:

```
def saveContact = Action { implicit request =>  
    contactForm.bindFromRequest.fold(  
        formWithErrors => {  
            BadRequest(views.html.contact.form(formWithErrors))  
,  
            contact => {  
                val contactId = Contact.save(contact)  
                Redirect(routes.Application.showContact(contactId)).flashing("success" -> "Contact saved!")  
            }  
)  
    })  
}
```

Next: Protecting against CSRF

Protecting against Cross Site Request Forgery

Cross Site Request Forgery (CSRF) is a security exploit where an attacker tricks a victim's browser into making a request using the victim's session. Since the session token is sent with every request, if an attacker can coerce the victim's browser to make a request on their behalf, the attacker can make requests on the user's behalf.

It is recommended that you familiarise yourself with CSRF, what the attack vectors are, and what the attack vectors are not. We recommend starting with [this information from OWASP](#).

Simply put, an attacker can coerce a victim's browser to make the following types of requests:

- All `GET` requests
- `POST` requests with bodies of type `application/x-www-form-urlencoded`, `multipart/form-data` and `text/plain`

An attacker can not:

- Coerce the browser to use other request methods such as `PUT` and `DELETE`
- Coerce the browser to post other content types, such as `application/json`
- Coerce the browser to send new cookies, other than those that the server has already set
- Coerce the browser to set arbitrary headers, other than the normal headers the browser adds to requests

Since `GET` requests are not meant to be mutative, there is no danger to an application that follows this best practice. So the only requests that need CSRF protection are `POST` requests with the above mentioned content types.

Play's CSRF protection

Play supports multiple methods for verifying that a request is not a CSRF request. The primary mechanism is a CSRF token. This token gets placed either in the query string or body of every form submitted, and also gets placed in the user's session. Play then verifies that both tokens are present and match.

To allow simple protection for non-browser requests, such as requests made through AJAX, Play also supports the following:

- If an `X-Requested-With` header is present, Play will consider the request safe. `X-Requested-With` is added to requests by many popular Javascript libraries, such as jQuery.
- If a `Csrf-Token` header with value `nocheck` is present, or with a valid CSRF token, Play will consider the request safe.

Applying a global CSRF filter

Play provides a global CSRF filter that can be applied to all requests. This is the simplest way to add CSRF protection to an application. To enable the global filter, add the Play filters helpers dependency to your project in `build.sbt`:

```
libraryDependencies += filters
```

Now add them to your `Filters` class as described in [HTTP filters](#):

```
import play.api.http.HttpFilters
import play.filters.csrf.CSRFFilter
```

```
import javax.inject.Inject

class Filters @Inject()(csrfFilter: CSRFFilter) extends HttpFilters {
  def filters = Seq(csrfFilter)
}
```

The `Filters` class can either be in the root package, or if it has another name or is in another package, needs to be configured using `play.http.filters` in `application.conf`:

```
play.http.filters = "filters.MyFilters"
```

Getting the current token

The current CSRF token can be accessed using the `getToken` method. It takes an implicit `RequestHeader`, so ensure that one is in scope.

```
import play.filters.csrf.CSRF
```

```
val token = CSRF.getToken(request)
```

To help in adding CSRF tokens to forms, Play provides some template helpers. The first one adds it to the query string of the action URL:

```
@import helper._

@form(CSRF(routes.ItemsController.save())) {
  ...
}
```

This might render a form that looks like this:

```
<form method="POST" action="/items?csrfToken=1234567890abcdef">
  ...
</form>
```

If it is undesirable to have the token in the query string, Play also provides a helper for adding the CSRF token as hidden field in the form:

```
@form(routes.ItemsController.save()) {
  @CSRF.formField
  ...
}
```

This might render a form that looks like this:

```
<form method="POST" action="/items">
  <input type="hidden" name="csrfToken" value="1234567890abcdef"/>
  ...
</form>
```

The form helper methods all require an implicit token or request to be available in scope. This will typically be provided by adding an implicit `RequestHeader` parameter to your template, if it doesn't have one already.

Adding a CSRF token to the session

To ensure that a CSRF token is available to be rendered in forms, and sent back to the client, the global filter will generate a new token for all GET requests that accept HTML, if a token isn't already available in the incoming request.

Applying CSRF filtering on a per action basis

Sometimes global CSRF filtering may not be appropriate, for example in situations where an application might want to allow some cross origin form posts. Some non session based standards, such as OpenID 2.0, require the use of cross site form posting, or use form submission in server to server RPC communications.

In these cases, Play provides two actions that can be composed with your applications actions.

The first action is the `CSRFCheck` action, and it performs the check. It should be added to all actions that accept session authenticated POST form submissions:

```
import play.api.mvc._  
import play.filters.csrf._
```

```
def save = CSRFCheck {  
    Action { req =>  
        // handle body  
        Ok  
    }  
}
```

The second action is the `CSRFAddToken` action, it generates a CSRF token if not already present on the incoming request. It should be added to all actions that render forms:

```
import play.api.mvc._  
import play.filters.csrf._
```

```
def form = CSRFAddToken {  
    Action { implicit req =>  
        Ok(views.html.itemsForm())  
    }  
}
```

A more convenient way to apply these actions is to use them in combination with Play's [action composition](#):

```
import play.api.mvc._  
import play.filters.csrf._
```

```

object PostAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) = CSRFCheck(action)
}

object GetAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) = CSRFAddToken(action)
}

```

Then you can minimise the boiler plate code necessary to write actions:

```

def save = PostAction {
  // handle body
  Ok
}

def form = GetAction { implicit req =>
  Ok(views.html.itemsForm())
}

```

CSRF configuration options

The full range of CSRF configuration options can be found in the filters [reference.conf](#). Some examples include:

- `play.filters.csrf.token.name` - The name of the token to use both in the session and in the request body/query string. Defaults to `csrfToken`.
- `play.filters.csrf.cookie.name` - If configured, Play will store the CSRF token in a cookie with the given name, instead of in the session.
- `play.filters.csrf.cookie.secure` - If `play.filters.csrf.cookie.name` is set, whether the CSRF cookie should have the secure flag set. Defaults to the same value as `play.http.session.secure`.
- `play.filters.csrf.body.bufferSize` - In order to read tokens out of the body, Play must first buffer the body and potentially parse it. This sets the maximum buffer size that will be used to buffer the body. Defaults to 100k.
- `play.filters.csrf.token.sign` - Whether Play should use signed CSRF tokens. Signed CSRF tokens ensure that the token value is randomised per request, thus defeating BREACH style attacks.

Next: [Custom Validations](#)

Using Custom Validations

The [validation package](#) allows you to create ad-hoc constraints using the `verifying` method. However, Play gives you the option of creating your own custom constraints, using the `Constraint` case class.

Here, we'll implement a simple password strength constraint that uses regular expressions to check the password is not all letters or all numbers. A `Constraint` takes a function which returns a `ValidationResult`, and we use that function to return the results of the password check:

```
val allNumbers = """^\d*""".r
val allLetters = """[A-Za-z]*""".r
```

```
val passwordCheckConstraint: Constraint[String] = Constraint("constraints.passwordcheck")({
  plainText =>
    val errors = plainText match {
      case allNumbers() => Seq(ValidationError("Password is all numbers"))
      case allLetters() => Seq(ValidationError("Password is all letters"))
      case _ => Nil
    }
    if (errors.isEmpty) {
      Valid
    } else {
      Invalid(errors)
    }
})
```

Note: This is an intentionally trivial example. Please consider using the [OWASP guide](#) for proper password security.

We can then use this constraint together with `Constraints.min` to add additional checks on the password.

```
val passwordCheck: Mapping[String] = nonEmptyText(minLength = 10)
  .verifying(passwordCheckConstraint)
```

Next: [Custom Field Constructors](#)

Custom Field Constructors

A field rendering is not only composed of the `<input>` tag, but it also needs a `<label>` and possibly other tags used by your CSS framework to decorate the field. All input helpers take an implicit `FieldConstructor` that handles this part. The [default one](#) (used if there are no other field constructors available in the scope), generates HTML like:

```
<dl class="error" id="username_field">
```

```

<dt><label for="username">Username:</label></dt>
<dd><input type="text" name="username" id="username" value=""></dd>
<dd class="error">This field is required!</dd>
<dd class="error">Another error</dd>
<dd class="info">Required</dd>
<dd class="info">Another constraint</dd>
</dl>

```

This default field constructor supports additional options you can pass in the input helper arguments:

```

'_label -> "Custom label"
'_id -> "idForTheTopDIElement"
'_help -> "Custom help"
'_showConstraints -> false
'_error -> "Force an error"
'_showErrors -> false

```

Writing your own field constructor

Often you will need to write your own field constructor. Start by writing a template like:

```

@(elements: helper.FieldElements)

<div class="@if(elements.hasErrors) { error }">
  <label for="@elements.id">@elements.label</label>
  <div class="input">
    @elements.input
    <span class="errors">@elements.errors.mkString(", ")</span>
    <span class="help">@elements.infos.mkString(", ")</span>
  </div>
</div>

```

Note: This is just a sample. You can make it as complicated as you need. You also have access to the original field using `@elements.field`.

Now create a `FieldConstructor` using this template function:

```

object MyHelpers {
  import views.html.helper.FieldConstructor
  implicit val myFields = FieldConstructor(html.myFieldConstructorTemplate.f)
}

```

And to make the form helpers use it, just import it in your templates:

```

@import MyHelpers._
@helper.inputText(myForm("username"))

```

It will then use your field constructor to render the input text.

You can also set an implicit value for your `FieldConstructor` inline:

```
@implicitField = @{} helper.FieldConstructor(myFieldConstructorTemplate.f) }  
@helper.inputText(myForm("username"))
```

Next: [Working with Json](#)

JSON basics

Modern web applications often need to parse and generate data in the JSON (JavaScript Object Notation) format. Play supports this via its [JSON library](#).

JSON is a lightweight data-interchange format and looks like this:

```
{  
  "name" : "Watership Down",  
  "location" : {  
    "lat" : 51.235685,  
    "long" : -1.309197  
  },  
  "residents" : [ {  
    "name" : "Fiver",  
    "age" : 4,  
    "role" : null  
  }, {  
    "name" : "Bigwig",  
    "age" : 6,  
    "role" : "Owsla"  
  } ]  
}
```

To learn more about JSON, see json.org.

The Play JSON library

The `play.api.libs.json` package contains data structures for representing JSON data and utilities for converting between these data structures and other data representations.

Types of interest are:

`JsValue`

This is a trait representing any JSON value. The JSON library has a case class extending `JsValue` to represent each valid JSON type:

- `JsString`
- `JsNumber`
- `JsBoolean`
- `JsObject`
- `JsArray`

- **JsNull**

Using the various `JsValue` types, you can construct a representation of any JSON structure.

Json

The `Json` object provides utilities, primarily for conversion to and from `JsValue` structures.

JsPath

Represents a path into a `JsValue` structure, analogous to XPath for XML. This is used for traversing `JsValue` structures and in patterns for implicit converters.

Converting to a `JsValue`

Using string parsing

```
import play.api.libs.json._

val json: JsValue = Json.parse("""
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
```

Using class construction

```
import play.api.libs.json._

val json: JsValue = JsObject(Seq(
  "name" -> JsString("Watership Down"),
  "location" -> JsObject(Seq("lat" -> JsNumber(51.235685), "long" -> JsNumber(-1.309197))),
  "residents" -> JsArray(Seq(
    JsObject(Seq(
      "name" -> JsString("Fiver"),
      "age" -> JsNumber(4),
      "role" -> JsNull
    )),
    JsObject(Seq(
      "name" -> JsString("Bigwig"),
      "age" -> JsNumber(6),
      "role" -> JsString("Owsla")
    ))
  ))
)
```

```
  ))
  ))
})
Json.obj and Json.arr can simplify construction a bit. Note that most values don't need
to be explicitly wrapped by JsValue classes, the factory methods use implicit conversion
(more on this below).
import play.api.libs.json.{JsNull,Json,JsString,JsValue}

val json: JsValue = Json.obj(
  "name" -> "Watership Down",
  "location" -> Json.obj("lat" -> 51.235685, "long" -> -1.309197),
  "residents" -> Json.arr(
    Json.obj(
      "name" -> "Fiver",
      "age" -> 4,
      "role" -> JsNull
    ),
    Json.obj(
      "name" -> "Bigwig",
      "age" -> 6,
      "role" -> "Owsla"
    )
  )
)
```

Using Writes converters

Scala to `JsValue` conversion is performed by the utility

method `Json.toJsonT(implicit writes: Writes[T])`. This functionality depends on a converter of type `Writes[T]` which can convert a `T` to a `JsValue`.

The Play JSON API provides implicit `Writes` for most basic types, such as

as `Int`, `Double`, `String`, and `Boolean`. It also supports `Writes` for collections of any type `T` that a `Writes[T]` exists.

```
import play.api.libs.json._
```

```
// basic types
val jsonString = Json.toJson("Fiver")
val jsonNumber = Json.toJson(4)
val jsonBoolean = Json.toJson(false)
```

```
// collections of basic types
val jsonArrayOfInts = Json.toJson(Seq(1, 2, 3, 4))
val jsonArrayOfStrings = Json.toJson(List("Fiver", "Bigwig"))
```

To convert your own models to `JsValue`s, you must define implicit `Writes` converters and provide them in scope.

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
import play.api.libs.json._
```

```
implicit val locationWrites = new Writes[Location] {
```

```

def writes(location: Location) = Json.obj(
  "lat" -> location.lat,
  "long" -> location.long
)
}

implicit val residentWrites = new Writes[Resident] {
  def writes(resident: Resident) = Json.obj(
    "name" -> resident.name,
    "age" -> resident.age,
    "role" -> resident.role
  )
}

implicit val placeWrites = new Writes[Place] {
  def writes(place: Place) = Json.obj(
    "name" -> place.name,
    "location" -> place.location,
    "residents" -> place.residents
}
}

val place = Place(
  "Watership Down",
  Location(51.235685, -1.309197),
  Seq(
    Resident("Fiver", 4, None),
    Resident("Bigwig", 6, Some("Owsla"))
  )
)

val json = Json.toJson(place)

```

Alternatively, you can define your `Writes` using the combinator pattern:

Note: The combinator pattern is covered in detail in [JSON Reads/Writes/Formats Combinators](#).
`import play.api.libs.json._`
`import play.api.libs.functional.syntax._`

```
implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))
```

```
implicit val residentWrites: Writes[Resident] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "age").write[Int] and
  (JsPath \ "role").writeNullable[String]
)(unlift(Resident.unapply))
```

```
implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location] and
  (JsPath \ "residents").write[Seq[Resident]]
)(unlift(Place.unapply))
```

Traversing a JsValue structure

You can traverse a `JsValue` structure and extract specific values. The syntax and functionality is similar to Scala XML processing.

Note: The following examples are applied to the JsValue structure created in previous examples.

Simple path

Applying the  operator to a `JsValue` will return the property corresponding to the field argument, supposing this is a `JsObject`.

```
val lat = (json \ "location" \ "lat").get  
// returns JsNumber(51.235685)
```

Recursive path

Applying the operator will do a lookup for the field in the current object and all descendants.

```
val names = json \\ "name"  
// returns Seq(JsString("Watership Down"), JsString("Fiver"), JsString("Bigwig"))
```

Index lookup (for JsArrays)

You can retrieve a value in a `JsArray` using an apply operator with the index number.

```
val bigwig = (json \ "residents")(1)  
// returns {"name": "Bigwig", "age": 6, "role": "Owsla"}
```

Converting from a JsValue

Using String utilities

Minified:

```
val minifiedString: String = Json.stringify(json)  
{ "name": "Watership Down", "location": { "lat": 51.235685, "long": -1.309197 }, "residents": [ { "name": "Fiver", "age": 4, "role": null }, { "name": "Bigwig", "age": 6, "role": "Owsla" } ] }
```

Readable:

```
val readableString: String = Json.prettyPrint(json)  
{  
  "name" : "Watership Down",  
  "location" : {  
    "lat" : 51.235685,  
    "long" : -1.309197  
  },  
  "residents" : [ {  
    "name" : "Fiver",  
    "age" : 4,  
    "role" : null  
  } ] }
```

```

    },
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  ]
}

```

Using JsValue.as/asOpt

The simplest way to convert a `JsValue` to another type is

using `JsValue.as[T]` (`implicit fjs: Reads[T]`) : `T`. This requires an implicit converter of type `Reads[T]` to convert a `JsValue` to `T` (the inverse of `Writes[T]`). As with `Writes`, the JSON API provides `Reads` for basic types.

```

val name = (json \ "name").as[String]
// "Watership Down"

```

```

val names = (json \\ "name").map(_.as[String])
// Seq("Watership Down", "Fiver", "Bigwig")

```

The `as` method will throw a `JsResultException` if the path is not found or the conversion is not possible. A safer method is `JsValue.asOpt[T]` (`implicit fjs: Reads[T]`) : `Option[T]`.

```

val nameOption = (json \ "name").asOpt[String]
// Some("Watership Down")

```

```

val bogusOption = (json \ "bogus").asOpt[String]
// None

```

Although the `asOpt` method is safer, any error information is lost.

Using validation

The preferred way to convert from a `JsValue` to another type is by using its `validate` method (which takes an argument of type `Reads`). This performs both validation and conversion, returning a type of `JsResult`. `JsResult` is implemented by two classes:

- `JsSuccess`: Represents a successful validation/conversion and wraps the result.
- `JsError`: Represents unsuccessful validation/conversion and contains a list of validation errors.

You can apply various patterns for handling a validation result:

```

val json = { ... }

val nameResult: JsResult[String] = (json \ "name").validate[String]

// Pattern matching
nameResult match {
  case s: JsSuccess[String] => println("Name: " + s.get)
  case e: JsError => println("Errors: " + JsError.toJson(e).toString())
}

// Fallback value
val nameOrFallback = nameResult.getOrElse("Undefined")

```

```
// map
val nameUpperResult: JsResult[String] = nameResult.map(_.toUpperCase())

// fold
val nameOption: Option[String] = nameResult.fold(
  invalid = {
    fieldErrors => fieldErrors.foreach(x => {
      println("field: " + x._1 + ", errors: " + x._2)
    })
    None
  },
  valid = {
    name => Some(name)
  }
)
```

JsValue to a model

To convert from JsValue to a model, you must define implicit `Reads[T]` where `T` is the type of your model.

Note: The pattern used to implement `Reads` and custom validation are covered in detail in [JSON Reads/Writes/Formats Combinators](#).

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)

implicit val residentReads: Reads[Resident] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "age").read[Int] and
  (JsPath \ "role").readNullable[String]
)(Resident.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location] and
  (JsPath \ "residents").read[Seq[Resident]]
)(Place.apply _)

val json = { ... }

val placeResult: JsResult[Place] = json.validate[Place]
// JsSuccess(Place(...),)

val residentResult: JsResult[Resident] = (json \ "residents")(1).validate[Resident]
// JsSuccess(Resident(Bigwig,6,Some(Owsla)),)
```

Next: [JSON with HTTP](#)

JSON with HTTP

Play supports HTTP requests and responses with a content type of JSON by using the HTTP API in combination with the JSON library.

See [HTTP Programming](#) for details on Controllers, Actions, and routing.

We'll demonstrate the necessary concepts by designing a simple RESTful web service to GET a list of entities and accept POSTs to create new entities. The service will use a content type of JSON for all data.

Here's the model we'll use for our service:

```
case class Location(lat: Double, long: Double)

case class Place(name: String, location: Location)

object Place {

    var list: List[Place] = {
        List(
            Place(
                "Sandleford",
                Location(51.377797, -1.318965)
            ),
            Place(
                "Watership Down",
                Location(51.235685, -1.309197)
            )
        )
    }

    def save(place: Place) = {
        list = list :: place
    }
}
```

Serving a list of entities in JSON

We'll start by adding the necessary imports to our controller.

```
import play.api.mvc._
import play.api.libs.json._
```

```

import play.api.libs.functional.syntax._

object Application extends Controller {

}

Before we write our Action, we'll need the plumbing for doing conversion from our model
to a JsValue representation. This is accomplished by defining an
implicit Writes[Place].
implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location]
)(unlift(Place.unapply))

```

Next we write our `Action`:

```

def listPlaces = Action {
  val json = Json.toJson(Place.list)
  Ok(json)
}

```

The `Action` retrieves a list of `Place` objects, converts them to a `JsValue` using `Json.toJson` with our implicit `Writes[Place]`, and returns this as the body of the result. Play will recognize the result as JSON and set the appropriate `Content-Type` header and body value for the response.

The last step is to add a route for our `Action` in `conf/routes`:

```

GET /places controllers.Application.listPlaces

```

We can test the action by making a request with a browser or HTTP tool. This example uses the unix command line tool [cURL](#).

```

curl --include http://localhost:9000/places

```

Response:

```

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 141

[{"name": "Sandleford", "location": {"lat": 51.377797, "long": -1.318965}}, {"name": "Watership
Down", "location": {"lat": 51.235685, "long": -1.309197}}]

```

Creating a new entity instance in JSON

For this `Action` we'll need to define an implicit `Reads[Place]` to convert a `JsValue` to our model.

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)
```

```
implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location]
)(Place.apply _)
```

Next we'll define the `Action`.

```
def savePlace = Action(BodyParsers.parse.json) { request =>
  val placeResult = request.body.validate[Place]
  placeResult.fold(
    errors => {
      BadRequest(Json.obj("status" ->"KO", "message" -> JsError.toJson(errors)))
    },
    place => {
      Place.save(place)
      Ok(Json.obj("status" ->"OK", "message" -> ("Place "+place.name+" saved.")))
    }
  )
}
```

This `Action` is more complicated than our list case. Some things to note:

- This `Action` expects a request with a `Content-Type` header of `text/json` or `application/json` and a body containing a JSON representation of the entity to create.
- It uses a JSON specific `BodyParser` which will parse the request and provide `request.body` as a `JsValue`.
- We used the `validate` method for conversion which will rely on our `implicit[Reads[Place]]`.
- To process the validation result, we used a `fold` with error and success flows. This pattern may be familiar as it is also used for [form submission](#).
- The `Action` also sends JSON responses.

Finally we'll add a route binding in `conf/routes`:

```
POST /places controllers.Application.savePlace
```

We'll test this action with valid and invalid requests to verify our success and error flows.

Testing the action with a valid data:

```
curl --include
--request POST
--header "Content-type: application/json"
--data '{ "name": "Nuthanger Farm", "location": { "lat" : 51.244031, "long" : -1.263224 } }'
http://localhost:9000/places
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 57
```

```
{"status":"OK","message":"Place 'Nuthanger Farm' saved."}
```

Testing the action with a invalid data, missing “name” field:

```
curl --include  
--request POST  
--header "Content-type: application/json"  
--data '{ "location":{ "lat" : 51.244031,"long" : -1.263224 } }'  
http://localhost:9000/places
```

Response:

```
HTTP/1.1 400 Bad Request  
Content-Type: application/json; charset=utf-8  
Content-Length: 79  
  
{ "status":"KO","message":{ "obj.name":[{ "msg":"error.path.missing","args":[] }]} }
```

Testing the action with a invalid data, wrong data type for “lat”:

```
curl --include  
--request POST  
--header "Content-type: application/json"  
--data '{ "name":"Nuthanger Farm","location":{ "lat" :"xxx","long" : -1.263224 } }'  
http://localhost:9000/places
```

Response:

```
HTTP/1.1 400 Bad Request  
Content-Type: application/json; charset=utf-8  
Content-Length: 92  
  
{ "status":"KO","message":{ "obj.location.lat":[{ "msg":"error.expected.jsnumber","args":[] }]} }
```

Summary

Play is designed to support REST with JSON and developing these services should hopefully be straightforward. The bulk of the work is in writing `Reads` and `Writes` for your model, which is covered in detail in the next section.

Next: [JSON Reads/Writes/Format Combinators](#)

JSON Reads/Writes/Format Combinators

JSON basics introduced `Reads` and `Writes` converters which are used to convert between `JsValue` structures and other data types. This page covers in greater detail how to build these converters and how to use validation during conversion.

The examples on this page will use this `JsValue` structure and corresponding model:

```
import play.api.libs.json._
```

```
val json: JsValue = Json.parse("""
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
```

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
```

JsPath

`JsPath` is a core building block for creating `Reads/Writes`. `JsPath` represents the location of data in a `JsValue` structure. You can use the `JsPath` object (root path) to define a `JsPath` child instance by using syntax similar to traversing `JsValue`:

```
import play.api.libs.json._
```

```
val json = { ... }

// Simple path
val latPath = JsPath \ "location" \ "lat"

// Recursive path
val namesPath = JsPath \\ "name"

// Indexed path
val firstResidentPath = (JsPath \ "residents")(0)

The play.api.libs.json package defines an alias for JsPath: __ (double underscore). You can use this if you prefer:
val longPath = __ \ "location" \ "long"
```

Reads

`Reads` converters are used to convert from a `JsValue` to another type. You can combine and nest `Reads` to create more complex `Reads`.

You will require these imports to create `Reads`:

```
import play.api.libs.json._ // JSON library
import play.api.libs.json.Reads._ // Custom validation helpers
import play.api.libs.functional.syntax._ // Combinator syntax
```

Path Reads

`JsPath` has methods to create special `Reads` that apply another `Reads` to a `JsValue` at a specified path:

- `JsPath.read[T](implicit r: Reads[T]): Reads[T]` - Creates a `Reads[T]` that will apply the implicit argument `r` to the `JsValue` at this path.
- `JsPath.readNullable[T](implicit r: Reads[T]): Reads[Option[T]]` - Use for paths that may be missing or can contain a null value.

Note: The JSON library provides implicit `Reads` for basic types such as `String`, `Int`, `Double`, etc.

Defining an individual path `Reads` looks like this:

```
val nameReads: Reads[String] = (JsPath \ "name").read[String]
```

Complex Reads

You can combine individual path `Reads` to form more complex `Reads` which can be used to convert to complex models.

For easier understanding, we'll break down the combine functionality into two statements.

First combine `Reads` objects using the `and` combinator:

```
val locationReadsBuilder =
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
```

This will yield a type of `FunctionalBuilder[Reads]#CanBuild2[Double, Double]`.

This is an intermediary object and you don't need to worry too much about it, just know that it's used to create a complex `Reads`.

Second call the `apply` method of `CanBuild2` with a function to translate individual values to your model, this will return your complex `Reads`. If you have a case class with a matching constructor signature, you can just use its `apply` method:

```
implicit val locationReads = locationReadsBuilder.apply(Location.apply _)
```

Here's the same code in a single statement:

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)
```

Validation with Reads

The `JsValue.validate` method was introduced in [JSON basics](#) as the preferred way to perform validation and conversion from a `JsValue` to another type. Here's the basic pattern:

```
val json = { ... }

val nameReads: Reads[String] = (JsPath \ "name").read[String]

val nameResult: JsResult[String] = json.validate[String](nameReads)

nameResult match {
  case s: JsSuccess[String] => println("Name: " + s.get)
  case e: JsError => println("Errors: " + JsError.toJson(e).toString())
}
```

Default validation for `Reads` is minimal, such as checking for type conversion errors. You can define custom validation rules by using `Reads` validation helpers. Here are some that are commonly used:

- `Reads.email` - Validates a String has email format.
- `Reads.minLength(nb)` - Validates the minimum length of a String.
- `Reads.min` - Validates a minimum numeric value.
- `Reads.max` - Validates a maximum numeric value.
- `Reads[A] keepAnd Reads[B] => Reads[A]` - Operator that tries `Reads[A]` and `Reads[B]` but only keeps the result of `Reads[A]` (For those who know Scala parser combinators `keepAnd == <~`).
- `Reads[A] andKeep Reads[B] => Reads[B]` - Operator that tries `Reads[A]` and `Reads[B]` but only keeps the result of `Reads[B]` (For those who know Scala parser combinators `andKeep == ~>`).
- `Reads[A] or Reads[B] => Reads` - Operator that performs a logical OR and keeps the result of the last `Reads` checked.

To add validation, apply helpers as arguments to the `JsPath.read` method:

```
val improvedNameReads =
  (JsPath \ "name").read[String](minLength[String](2))
```

Putting it all together

By using complex `Reads` and custom validation we can define a set of effective `Reads` for our example model and apply them:

```
import play.api.libs.json._
import play.api.libs.json.Reads._
import play.api.libs.functional.syntax._

implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply _)

implicit val residentReads: Reads[Resident] = (
  (JsPath \ "name").read[String](minLength[String](2)) and
  (JsPath \ "age").read[Int](min(0) keepAnd max(150)) and
```

```

(JsPath \ "role").readNullable[String]
)(Resident.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String](minLength[String](2)) and
  (JsPath \ "location").read[Location] and
  (JsPath \ "residents").read[Seq[Resident]]
)(Place.apply _)

val json = { ... }

json.validate[Place] match {
  case s: JsSuccess[Place] => {
    val place: Place = s.get
    // do something with place
  }
  case e: JsError => {
    // error handling flow
  }
}

```

Note that complex `Reads` can be nested. In this case, `placeReads` uses the previously defined implicit `locationReads` and `residentReads` at specific paths in the structure.

Writes

`Writes` converters are used to convert from some type to a `JsValue`.

You can build complex `Writes` using `JsPath` and combinators very similar to `Reads`.

Here's the `Writes` for our example model:

```

import play.api.libs.json._  
import play.api.libs.functional.syntax._

```

```

implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

```

```

implicit val residentWrites: Writes[Resident] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "age").write[Int] and
  (JsPath \ "role").writeNullable[String]
)(unlift(Resident.unapply))

```

```

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location] and
  (JsPath \ "residents").write[Seq[Resident]]
)(unlift(Place.unapply))

```

```
val place = Place(
```

```

    "Watership Down",
    Location(51.235685, -1.309197),
    Seq(
      Resident("Fiver", 4, None),
      Resident("Bigwig", 6, Some("Owsla"))
    )
  )

  val json = Json.toJson(place)

```

There are a few differences between complex `Writes` and `Reads`:

- The individual path `Writes` are created using the `JsPath.write` method.
- There is no validation on conversion to `JsValue` which makes the structure simpler and you won't need any validation helpers.
- The intermediary `FunctionalBuilder#CanBuildX` (created by `and` combinators) takes a function that translates a complex type `T` to a tuple matching the individual path `Writes`. Although this is symmetrical to the `Reads` case, the `unapply` method of a case class returns an `Option` of a tuple of properties and must be used with `unlift` to extract the tuple.

Recursive Types

One special case that our example model doesn't demonstrate is how to handle `Reads` and `Writes` for recursive

types. `JsPath` provides `lazyRead` and `lazyWrite` methods that take call-by-name parameters to handle this:

```
case class User(name: String, friends: Seq[User])
```

```
implicit lazy val userReads: Reads[User] = (
  __ \ "name").read[String] and
  __ \ "friends").lazyRead(Reads.seq[User](userReads))
)(User)
```

```
implicit lazy val userWrites: Writes[User] = (
  __ \ "name").write[String] and
  __ \ "friends").lazyWrite(Writes.seq[User](userWrites))
)(unlift(User.unapply))
```

Format

`Format[T]` is just a mix of the `Reads` and `Writes` traits and can be used for implicit conversion in place of its components.

Creating Format from Reads and Writes

You can define a `Format` by constructing it from `Reads` and `Writes` of the same type:

```
val locationReads: Reads[Location] = (
  JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply _)
```

```

val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val locationFormat: Format[Location] =
  Format(locationReads, locationWrites)

```

Creating Format using combinators

In the case where your `Reads` and `Writes` are symmetrical (which may not be the case in real applications), you can define a `Format` directly from combinators:

```

implicit val locationFormat: Format[Location] = (
  (JsPath \ "lat").format[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").format[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply, unlift(Location.unapply))

```

[Next: JSON Transformers](#)

JSON transformers

Please note this documentation was initially published as an article by Pascal Voitot (@mandubian) on mandubian.com

Now you should know how to validate JSON and convert into any structure you can write in Scala and back to JSON. But as soon as I've begun to use those combinators to write web applications, I almost immediately encountered a case : read JSON from network, validate it and convert it into... JSON.

Introducing JSON *coast-to-coast* design

Are we doomed to convert JSON to OO?

For a few years now, in almost all web frameworks (except recent JavaScript server side stuff maybe in which JSON is the default data structure), we have been used to get JSON from network and **convert JSON (or even POST/GET data) into OO structures** such as classes (or case classes in Scala). Why?

- For a good reason : **OO structures are “language-native”** and allows **manipulating data with respect to your business logic** in a seamless way while ensuring isolation of business logic from web layers.
- For a more questionable reason : **ORM frameworks talk to DB only with OO structures** and we have (kind of) convinced ourselves that it was impossible to do else... with the well-known good & bad features of ORMs... (not here to criticize those stuff)

Is OO conversion really the default use case?

In many cases, you don't really need to perform any real business logic with data but validating/transforming before storing or after extracting. Let's take the CRUD case:

- You just get the data from the network, validate them a bit and insert/update into DB.
- In the other way, you just retrieve data from DB and send them outside.

So, generally, for CRUD ops, you convert JSON into a OO structure just because the frameworks are only able to speak OO.

I don't say or pretend you shouldn't use JSON to OO conversion but maybe this is not the most common case and we should keep conversion to OO only when we have real business logic to fulfill.

New tech players change the way of manipulating JSON

Besides this fact, we have some new DB types such as MongoDB (or CouchDB) accepting document structured data looking almost like JSON trees (_isn't BSON, Binary JSON?_).

With these DB types, we also have new great tools such as [ReactiveMongo](#) which provides reactive environment to stream data to and from Mongo in a very natural way.

I've been working with Stephane Godbillon to integrate ReactiveMongo with Play2.1 while writing the [Play2-ReactiveMongo module](#). Besides Mongo facilities for Play2.1, this module provides *Json To/From BSON conversion typeclasses*.

So it means you can manipulate JSON flows to and from DB directly without even converting into OO.

JSON *coast-to-coast* design

Taking this into account, we can easily imagine the following:

- Receive JSON.
- Validate JSON.
- Transform JSON to fit expected DB document structure.
- Directly send JSON to DB (or somewhere else).

This is exactly the same case when serving data from DB:

- Extract some data from DB as JSON directly.
- Filter/transform this JSON to send only mandatory data in the format expected by the client (e.g you don't want some secure info to go out).
- Directly send JSON to the client.

In this context, we can easily imagine **manipulating a flow of JSON data** from client to DB and back without any (explicit) transformation in anything else than JSON.

Naturally, when you plug this transformation flow on **reactive infrastructure provided by Play2.1**, it suddenly opens new horizons.

This is the so-called (by me) **JSON coast-to-coast design**:

- Don't consider JSON data chunk by chunk but as a **continuous flow of data from client to DB (or else) through server**,
- Treat the **JSON flow like a pipe that you connect to others pipes** while applying modifications, transformations alongside,
- Treat the flow in a **fully asynchronous/non-blocking** way.

This is also one of the reason of being of Play2.1 reactive architecture...

I believe **considering your app through the prism of flows of data changes drastically the way you design** your web apps in general. It may also open new functional scopes that fit today's webapps requirements quite better than classic architecture. Anyway, this is not the subject here ;)

So, as you have deduced by yourself, to be able to manipulate Json flows based on validation and transformation directly, we needed some new tools. JSON combinators were good candidates but they are a bit too generic.

That's why we have created some specialized combinators and API called **JSON transformers** to do that.

JSON transformers

are Reads [T <: JsValue]

- You may tell JSON transformers are just `f: JSON => JSON`.
- So a JSON transformer could be simply a `Writes[A <: JsValue]`.
- But, a JSON transformer is not only a function: as we said, we also want to validate JSON while transforming it.
- As a consequence, a JSON transformer is a `Reads[A <: JsValue]`.

Keep in mind that a `Reads[A <: JsValue]` is able to transform and not only to read/validate

Use `JsValue.transform` instead of `JsValue.validate`

We have provided a function helper in `JsValue` to help people consider a `Reads[T]` is a transformer and not only a validator:

`JsValue.transform[A <: JsValue](reads: Reads[A]): JsResult[A]`

This is exactly the same `JsValue.validate(reads)`

The details

In the code samples below, we'll use the following JSON:

```
{  
  "key1" : "value1",  
  "key2" : {  
    "key21" : 123,  
    "key22" : true,  
    "key23" : [ "alpha", "beta", "gamma"],  
    "key24" : {  
      "key241" : 234.123,  
      "key242" : "value242"  
    }  
  },  
  "key3" : 234  
}
```

Case 1: Pick JSON value in JsPath

Pick value as JsValue

```
import play.api.libs.json._  
  
val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick  
  
scala> json.transform(jsonTransformer)  
res9: play.api.libs.json.JsResult[play.api.libs.json.JsValue] =  
  JsSuccess(  
    ["alpha", "beta", "gamma"],  
    /key2/key23  
  )  
  (__ \ 'key2 \ 'key23).json...
```

- All JSON transformers are in `JsPath.json`.
`(__ \ 'key2 \ 'key23).json.pick`
- `pick` is a `Reads[JsValue]` which picks the value **IN** the given JsPath.
Here `["alpha", "beta", "gamma"]`
`JsSuccess(["alpha", "beta", "gamma"], /key2/key23)`
- This is a simply successful `JsResult`.
- For info, `/key2/key23` represents the JsPath where data were read but don't care about it, it's mainly used by Play API to compose `JsResult(s)`.
- `["alpha", "beta", "gamma"]` is just due to the fact that we have overridden `toString`.

Reminder

`jsPath.json.pick` gets ONLY the value inside the JsPath

Pick value as Type

```
import play.api.libs.json._  
  
val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick[JsArray]
```

- ```
scala> json.transform(jsonTransformer)
res10: play.api.libs.json.JsResult[play.api.libs.json.JsArray] =
 JsSuccess(
 ["alpha", "beta", "gamma"],
 /key2/key23
)
 (__ \ 'key2 \ 'key23).json.pick[JsArray]
```
- `pick[T]` is a `Reads[T <: JsValue]` which picks the value (as a `JsArray` in our case) IN the given `JsPath`
- Reminder**
- `jsPath.json.pick[T <: JsValue]` extracts ONLY the typed value inside the `JsPath`

## Case 2: Pick branch following `JsPath`

Pick branch as `JsValue`

```
import play.api.libs.json._
```

```
val jsonTransformer = (__ \ 'key2 \ 'key24 \ 'key241).json.pickBranch
```

```
scala> json.transform(jsonTransformer)
res11: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
 "key2": {
 "key24": {
 "key241": 234.123
 }
 }
 },
 /key2/key24/key241
)
```

- `pickBranch` is a `Reads[JsValue]` which picks the branch from root to given `JsPath`  
`{"key2": {"key24": {"key241": "value241" }}}}`
- The result is the branch from root to given `JsPath` including the `JsValue` in `JsPath`

**Reminder:**

`jsPath.json.pickBranch` extracts the single branch down to `JsPath` + the value inside `JsPath`

## Case 3: Copy a value from input `JsPath` into a new `JsPath`

```
import play.api.libs.json._
```

```

val jsonTransformer = (__ \ 'key25 \ 'key251).json.copyFrom(__ \ 'key2 \ 'key21).json.pick)

scala> json.transform(jsonTransformer)
res12: play.api.libs.json.JsResult[play.api.libs.json.JsObject]
JsSuccess(
 {
 "key25":{
 "key251":123
 }
 },
 /key2/key21
)
(__ \ 'key25 \ 'key251).json.copyFrom(reads: Reads[A <: JsValue])

```

- `copyFrom` is a `Reads[JsValue]`
- `copyFrom` reads the `JsValue` from input JSON using provided `Reads[A]`
- `copyFrom` copies this extracted `JsValue` as the leaf of a new branch corresponding to given `JsPath`  
`{"key25": {"key251": 123}}`
- `copyFrom` reads value `123`
- `copyFrom` copies this value into new branch `( __ \ 'key25 \ 'key251)`

#### Reminder:

`jsPath.json.copyFrom(Reads[A <: JsValue])` reads value from input JSON and creates a new branch with result as leaf

## Case 4: Copy full input Json & update a branch

```

import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key24).json.update(
 __.read[JsObject].map{ o => o ++ Json.obj("field243" -> "coucou") }
)

scala> json.transform(jsonTransformer)
res13: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
JsSuccess(
 {
 "key1": "value1",
 "key2": {
 "key21": 123,
 "key22": true,
 "key23": ["alpha", "beta", "gamma"],
 "key24": {
 "key241": 234.123,
 "key242": "value242",
 "field243": "coucou"
 }
 }
 }
)

```

```

 }
 },
 "key3":234
},
)
(_ \ 'key2).json.update(reads: Reads[A < JsValue])

```

- Is a `Reads[JsObject]`  
`(_ \ 'key2 \ 'key24).json.update(reads)` does 3 things:
- Extracts value from input JSON at JsPath `(_ \ 'key2 \ 'key24)`.
- Applies `reads` on this relative value and re-creates a branch `(_ \ 'key2 \ 'key24)` adding result of `reads` as leaf.
- Merges this branch with full input JSON replacing existing branch (so it works only with input `JsObject` and not other type of `JsValue`).  
`JsSuccess(...)`
- Just for info, there is no JsPath as 2nd parameter there because the JSON manipulation was done from Root JsPath

**Reminder:**

`jsPath.json.update(Reads[A <: JsValue])` only works for `JsObject`, copies full input `JsObject` and updates jsPath with provided `Reads[A <: JsValue]`

## Case 5: Put a given value in a new branch

```

import play.api.libs.json._

val jsonTransformer = (_ \ 'key24 \ 'key241).json.put(JsNumber(456))

scala> json.transform(jsonTransformer)
res14: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
 "key24": {
 "key241": 456
 }
 },
)
(_ \ 'key24 \ 'key241).json.put(a: => JsValue)

```

- Is a `Reads[JsObject]`  
`(_ \ 'key24 \ 'key241).json.put(a: => JsValue)`
- Creates a new branch `(_ \ 'key24 \ 'key241)`
- Puts `a` as leaf of this branch.  
`jsPath.json.put(a: => JsValue)`
- Takes a `JsValue` argument passed by name allowing to pass even a closure to it.  
`jsPath.json.put`
- Does not care at all about input JSON.

- Simply replace input JSON by given value.

\*\*Reminder: \*\*

`jsPath.json.put( a: => Jsvalue )` creates a new branch with a given value without taking into account input JSON

## Case 6: Prune a branch from input JSON

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key22).json.prune

scala> json.transform(jsonTransformer)
res15: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
 "key1": "value1",
 "key3": 234,
 "key2": {
 "key21": 123,
 "key23": ["alpha", "beta", "gamma"],
 "key24": {
 "key241": 234.123,
 "key242": "value242"
 }
 },
 "/key2/key22/key22"
 }
 \ 'key2 \ 'key22).json.prune
```

- Is a `Reads[JsObject]` that works only with `JsObject`

`(__ \ 'key2 \ 'key22).json.prune`

- Removes given `JsPath` from input JSON (`key22` has disappeared under `key2`)

Please note the resulting `JsObject` hasn't same keys order as input `JsObject`. This is due to the implementation of `JsObject` and to the merge mechanism. But this is not important since we have overridden `JsObject.equals` method to take this into account.

**Reminder:**

`jsPath.json.prune` only works with `JsObject` and removes given `JsPath` from input JSON)

Please note that:

- `prune` doesn't work for recursive `JsPath` for the time being

- if `prune` doesn't find any branch to delete, it doesn't generate any error and returns unchanged JSON.

## More complicated cases

# Case 7: Pick a branch and update its content in 2 places

```
import play.api.libs.json._
import play.api.libs.json.Reads._

val jsonTransformer = (_ \ 'key2).json.pickBranch(
 (_ \ 'key21).json.update(
 of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }
) andThen
 (_ \ 'key23).json.update(
 of[JsArray].map{ case JsArray(arr) => JsArray(arr :+ JsString("delta")) }
)
)

scala> json.transform(jsonTransformer)
res16: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
 "key2": {
 "key21": 133,
 "key22": true,
 "key23": ["alpha", "beta", "gamma", "delta"],
 "key24": {
 "key241": 234.123,
 "key242": "value242"
 }
 },
 /key2
 })
 (_ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])
```

- Extracts branch `_ \ 'key2` from input JSON and applies `reads` to the relative leaf of this branch (only to the content).  
`(_ \ 'key21).json.update(reads: Reads[A <: JsValue])`
- Updates `(_ \ 'key21)` branch.  
`of[JsNumber]`
- Is just a `Reads[JsNumber]`.
- Extracts a JsNumber from `(_ \ 'key21)`.  
`of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }`
- Reads a JsNumber (`_value 123_` in `_ \ 'key21`).
- Uses `Reads[A].map` to increase it by `10` (in immutable way naturally).  
`andThen`
- Is just the composition of 2 `Reads[A]`.
- First reads is applied and then result is piped to second reads.

```
of[JsArray].map{ case JsArray(arr) => JsArray(arr :+
JsString("delta"))}
```

- Reads a JsArray (\_value [alpha, beta, gamma] in `__ \ 'key2`) in `__ \ 'key23`).
- Uses `Reads[A].map` to append `JsString("delta")` to it.

Please note the result is just the `__ \ 'key2` branch since we picked only this branch

## Case 8: Pick a branch and prune a sub-branch

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2).json.pickBranch(
 (__ \ 'key23).json.prune
)

scala> json.transform(jsonTransformer)
res18: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
 "key2": {
 "key21": 123,
 "key22": true,
 "key24": {
 "key241": 234.123,
 "key242": "value242"
 }
 },
 /key2/key23
 })
 (__ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])
```

- Extracts branch `__ \ 'key2` from input JSON and applies `reads` to the relative leaf of this branch (only to the content).

```
(__ \ 'key23).json.prune
```

- Removes branch `__ \ 'key23` from relative JSON

Please remark the result is just the `__ \ 'key2` branch without `key23` field.

## What about combinators?

I stop there before it becomes boring (if not yet)...

Just keep in mind that you have now a huge toolkit to create generic JSON transformers. You can compose, map, flatmap transformers together into other transformers. So possibilities are almost infinite.

But there is a final point to treat: mixing those great new JSON transformers with previously presented Reads combinators. This is quite trivial as JSON transformers are just `Reads[A <: JsValue]`

Let's demonstrate by writing a **Gizmo to Gremlin** JSON transformer.

Here is Gizmo:

```
val gizmo = Json.obj(
 "name" -> "gizmo",
 "description" -> Json.obj(
 "features" -> Json.arr("hairy", "cute", "gentle"),
 "size" -> 10,
 "sex" -> "undefined",
 "life_expectancy" -> "very old",
 "danger" -> Json.obj(
 "wet" -> "multiplies",
 "feed after midnight" -> "becomes gremlin"
)
),
 "loves" -> "all"
)
```

Here is Gremlin:

```
val gremlin = Json.obj(
 "name" -> "gremlin",
 "description" -> Json.obj(
 "features" -> Json.arr("skinny", "ugly", "evil"),
 "size" -> 30,
 "sex" -> "undefined",
 "life_expectancy" -> "very old",
 "danger" -> "always"
),
 "hates" -> "all"
)
```

Ok let's write a JSON transformer to do this transformation

```
import play.api.libs.json._
import play.api.libs.json.Reads._
import play.api.libs.functional.syntax._

val gizmo2gremlin = (
 __ \ 'name).json.put(JsString("gremlin")) and
 __ \ 'description).json.pickBranch(
 __ \ 'size).json.update(of[JsNumber].map{ case JsNumber(size) => JsNumber(size * 3) }) and
 __ \ 'features).json.put(Json.arr("skinny", "ugly", "evil")) and
 __ \ 'danger).json.put(JsString("always"))
 reduce
) and
(__ \ 'hates).json.copyFrom(__ \ 'loves).json.pick()
```

```
) reduce

scala> gizmo.transform(gizmo2gremlin)
res22: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
JsSuccess(
{
 "name": "gremlin",
 "description": {
 "features": ["skinny", "ugly", "evil"],
 "size": 30,
 "sex": "undefined",
 "life_expectancy": "very old",
 "danger": "always"
 },
 "hates": "all"
},
)
```

Here we are ;)

I'm not going to explain all of this because you should be able to understand now.

Just remark:

`(__ \ 'features).json.put(...)` is after `(__ \ 'size).json.update` so  
*that it overwrites original* `(__ \ 'features)`

`(Reads[JsObject] and Reads[JsObject]) reduce`

- It merges results of both `Reads[JsObject]` (`JsObject ++ JsObject`)
- It also applies the same JSON to both `Reads[JsObject]` unlike `andThen` which injects the result of the first reads into second one.

**Next: JSON Macro Inception**

# JSON Macro Inception

Please note this documentation was initially published as an article by Pascal Voitot (@mandubian) on [mandubian.com](http://mandubian.com)

This feature is still experimental because Scala Macros are still experimental in Scala 2.10. If you prefer not using an experimental feature from Scala, please use hand-written `Reads/Writes/Format` which are strictly equivalent.

# Writing a default case class Reads/Writes/Format is so boring!

Remember how you write a `Reads[T]` for a case class.

```

import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = (
 __ \ 'name).read[String] and
 __ \ 'age).read[Int] and
 __ \ 'lovesChocolate).read[Boolean]
)(Person)

```

So you write 4 lines for this case class.

You know what?

We have had a few complaints from some people who think it's not cool to write `aReads[TheirClass]` because usually Java JSON frameworks like Jackson or Gson do it behind the curtain without writing anything.

We argued that Play2.1 JSON serializers/deserializers are:

- completely typesafe,
- fully compiled,
- nothing was performed using introspection/reflection at runtime.

But for some, this didn't justify the extra lines of code for case classes.

We believe this is a really good approach so we persisted and proposed:

- JSON simplified syntax
- JSON combinators
- JSON transformers

Added power, but nothing changed for the additional 4 lines.

## Let's be minimalist

As we are perfectionist, now we propose a new way of writing the same code:

```

import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = Json.reads[Person]

```

1 line only.

Questions you may ask immediately:

Does it use runtime bytecode enhancement? -> NO

Does it use runtime introspection? -> NO

Does it break type-safety? -> NO

**So what?**

After creating buzzword **JSON coast-to-coast design**, let's call it **JSON INCEPTION**.

# JSON Inception

## Code Equivalence

As explained just before:

```
import play.api.libs.json._
// please note we don't import functional.syntax._ as it is managed by the macro itself

implicit val personReads = Json.reads[Person]

// IS STRICTLY EQUIVALENT TO writing

implicit val personReads = (
 __ \ 'name).read[String] and
 __ \ 'age).read[Int] and
 __ \ 'lovesChocolate).read[Boolean]
(Person)
```

## Inception equation

Here is the equation describing the windy *Inception* concept:

(Case Class INSPECTION) + (Code INJECTION) + (COMPILE Time) = INCEPTION

### Case Class Inspection

As you may deduce by yourself, in order to ensure preceding code equivalence, we need :

- to inspect `Person` case class,
- to extract the 3 fields `name`, `age`, `lovesChocolate` and their types,
- to resolve typeclasses implicits,

- to find `Person.apply`.

## *INJECTION?*

No I stop you immediately...

**Code injection is not dependency injection...**

No Spring behind inception... No IOC, No DI... No No No ;)

I used this term on purpose because I know that injection is now linked immediately to IOC and Spring. But I'd like to re-establish this word with its real meaning.

Here code injection just means that we **inject code at compile-time into the compiled scala AST (Abstract Syntax Tree)**.

So `Json.reads[Person]` is compiled and replaced in the compile AST by:

```
(
 __ \ 'name).read[String] and
 __ \ 'age).read[Int] and
 __ \ 'lovesChocolate).read[Boolean]
(Person)
```

Nothing less, nothing more...

## *COMPILE-TIME*

Yes everything is performed at compile-time.

No runtime bytecode enhancement.

No runtime introspection.

As everything is resolved at compile-time, you will have a compile error if you did not import the required implicits for all the types of the fields.

# Json inception is Scala 2.10 Macros

We needed a Scala feature enabling:

- compile-time code enhancement
- compile-time class/implicits inspection

- compile-time code injection

This is enabled by a new experimental feature introduced in Scala 2.10: **Scala Macros**

Scala macros is a new feature (still experimental) with a huge potential. You can :

- introspect code at compile-time based on Scala reflection API,
- access all imports, implicits in the current compile context
- create new code expressions, generate compiling errors and inject them into compile chain.

Please note that:

- **We use Scala Macros because it corresponds exactly to our requirements.**
- **We use Scala macros as an enabler, not as an end in itself.**
- **The macro is a helper that generates the code you could write by yourself.**
- **It doesn't add, hide unexpected code behind the curtain.**
- **We follow the *no-surprise* principle**

As you may discover, writing a macro is not a trivial process since your macro code executes in the compiler runtime (or universe).

So you write macro code  
 that **is** compiled **and** executed  
**in** a runtime that manipulates your code  
 to be compiled **and** executed  
**in** a future runtime...

That's also certainly why I called it *Inception* ;)

So it requires some mental exercises to follow exactly what you do. The API is also quite complex and not fully documented yet. Therefore, you must persevere when you begin using macros.

I'll certainly write other articles about Scala macros because there are lots of things to say. This article is also meant to **begin the reflection about the right way to use Scala Macros**. Great power means greater responsibility so it's better to discuss all together and establish a few good manners...

## Writes[T] & Format[T]

Please remark that JSON inception just works for structures having `unapply/apply` functions with corresponding input/output types.

Naturally, you can also `incept Writes[T]` and `Format[T]`.

## Writes[T]

```
import play.api.libs.json._

implicit val personWrites = Json.writes[Person]
```

## Format[T]

```
import play.api.libs.json._

implicit val personWrites = Json.format[Person]
```

## Special patterns

- You can define your Reads/Writes in your companion object

This is useful because then the implicit Reads/Writes is implicitly inferred as soon as you manipulate an instance of your class.

```
import play.api.libs.json._
```

```
case class Person(name: String, age: Int)

object Person{
 implicit val personFmt = Json.format[Person]
}
```

- You can now define Reads/Writes for single-field case class (known limitation until 2.1-RC2)

```
import play.api.libs.json._
```

```
case class Person(names: List[String])

object Person{
 implicit val personFmt = Json.format[Person]
}
```

## Known limitations

- Don't override apply function in companion object because then the Macro will have several apply functions and won't choose.
- Json Macros only work when apply and unapply have corresponding input/output types: This is naturally the case for case classes. But if you want the same with a trait, you must implement the same apply/unapply you would have in a case class.
- Json Macros are known to accept Option/Seq/List/Set & Map[String, \_]. For other generic types, test and if not working, use traditional way of writing Reads/Writes manually.

Next: Working with XML

# Handling and serving XML requests

## Handling an XML request

An XML request is an HTTP request using a valid XML payload as the request body. It must specify the `application/xml` or `text/xml` MIME type in its `Content-Type` header.

By default an `Action` uses a **any content** body parser, which lets you retrieve the body as XML (actually as a `NodeSeq`):

```
def sayHello = Action { request =>
 request.body.asXml.map { xml =>
 (xml \\"name" headOption).map(_.text).map { name =>
 Ok("Hello " + name)
 }.getOrElse {
 BadRequest("Missing parameter [name]")
 }
 }.getOrElse {
 BadRequest("Expecting Xml data")
 }
}
```

It's way better (and simpler) to specify our own `BodyParser` to ask Play to parse the content body directly as XML:

```
def sayHello = Action(parse.xml) { request =>
 (request.body \\"name" headOption).map(_.text).map { name =>
 Ok("Hello " + name)
 }.getOrElse {
 BadRequest("Missing parameter [name]")
 }
}
```

**Note:** When using an XML body parser, the `request.body` value is directly a valid `NodeSeq`.

You can test it with `cURL` from a command line:

```
curl
--header "Content-type: application/xml"
--request POST
--data '<name>Guillaume</name>'
http://localhost:9000/sayHello
```

It replies with:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 15
```

# Serving an XML response

In our previous example we handle an XML request, but we reply with a `text/plain` response. Let's change that to send back a valid XML HTTP response:

```
def sayHello = Action(parse.xml) { request =>
 (request.body \\"name" headOption).map(_.text).map { name =>
 Ok(<message status="OK">Hello {name}</message>)
 }.getOrElse {
 BadRequest(<message status="KO">Missing parameter [name]</message>)
 }
}
```

Now it replies with:

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
Content-Length: 46

<message status="OK">Hello Guillaume</message>
Next: Handling file upload
```

# Handling file upload

## Uploading files in a form using multipart/form-data

The standard way to upload files in a web application is to use a form with a special `multipart/form-data` encoding, which lets you mix standard form data with file attachment data.

**Note:** The HTTP method used to submit the form must be `POST` (not `GET`).

Start by writing an HTML form:

```
@helper.form(action = routes.Application.upload, 'enctype -> "multipart/form-data") {

 <input type="file" name="picture">

 <p>
 <input type="submit">
 </p>
}
```

```

}

Now define the upload action using a multipartFormData body parser:
def upload = Action(parse.multipartFormData) { request =>
 request.body.file("picture").map { picture =>
 import java.io.File
 val filename = picture.filename
 val contentType = picture.contentType
 picture.ref.moveTo(new File(s"/tmp/picture/$filename"))
 Ok("File uploaded")
 }.getOrElse {
 Redirect(routes.Application.index).flashing(
 "error" -> "Missing file")
 }
}

```

The `ref` attribute give you a reference to a `TemporaryFile`. This is the default way the `multipartFormData` parser handles file upload.

**Note:** As always, you can also use the `anyContent` body parser and retrieve it as `request.body.asMultipartFormData`.

At last, add a `POST` router

```
POST / controllers.Application.upload()
```

## Direct file upload

Another way to send files to the server is to use Ajax to upload the file asynchronously in a form. In this case the request body will not have been encoded as `multipart/form-data`, but will just contain the plain file content.

In this case we can just use a body parser to store the request body content in a file. For this example, let's use the `temporaryFile` body parser:

```

def upload = Action(parse.temporaryFile) { request =>
 request.body.moveTo(new File("/tmp/picture/uploaded"))
 Ok("File uploaded")
}

```

## Writing your own body parser

If you want to handle the file upload directly without buffering it in a temporary file, you can just write your own `BodyParser`. In this case, you will receive chunks of data that you are free to push anywhere you want.

If you want to use `multipart/form-data` encoding, you can still use the default `multipartFormData` parser by providing your own `PartHandler[FilePart[A]]`. You receive the part headers, and you have to provide an `Iteratee[Array[Byte], FilePart[A]]` that will produce the right `FilePart`.

**Next:** Accessing an SQL database

# Accessing an SQL database

---

## Configuring JDBC connection pools

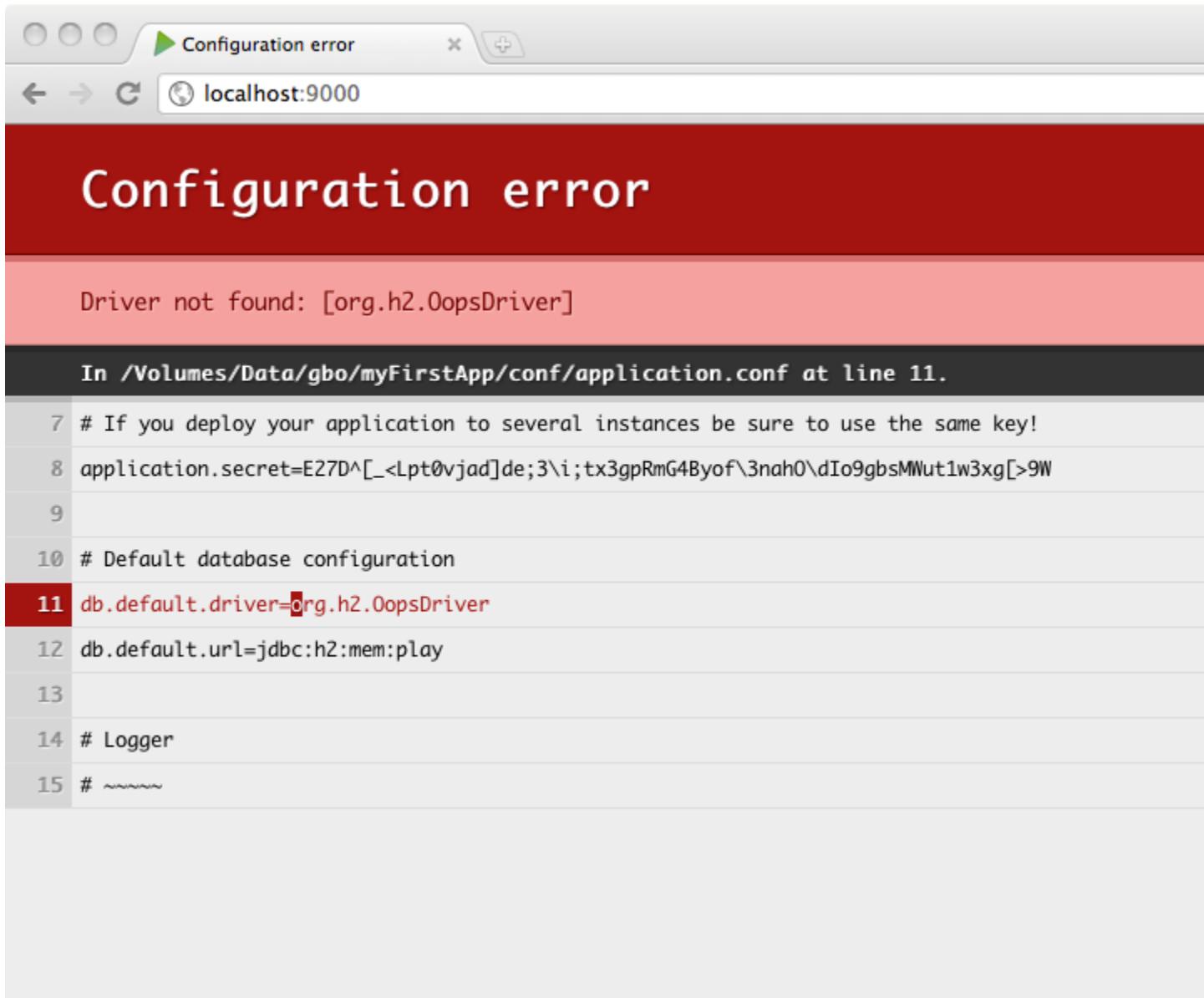
Play provides a plug-in for managing JDBC connection pools. You can configure as many databases as you need.

To enable the database plug-in, add jdbc in your build dependencies :

```
libraryDependencies += jdbc
```

Then you must configure a connection pool in the `conf/application.conf` file. By convention, the default JDBC data source must be called `default` and the corresponding configuration properties are `db.default.driver` and `db.default.url`.

If something isn't properly configured you will be notified directly in your browser:



The screenshot shows a web browser window with the title "Configuration error". The address bar indicates the URL is "localhost:9000". The main content area has a red header with the text "Configuration error". Below this, a pink section displays the error message "Driver not found: [org.h2.OopsDriver]". The subsequent text is a code snippet from a configuration file:

```
In /Volumes/Data/gbo/myFirstApp/conf/application.conf at line 11.
7 # If you deploy your application to several instances be sure to use the same key!
8 application.secret=E27D^[_<Lpt0vjad]de;3\i;tx3gpRmG4Byof\3nah0\dIo9gbsMNut1w3xg[>9W
9
10 # Default database configuration
11 db.default.driver=org.h2.OopsDriver
12 db.default.url=jdbc:h2:mem:play
13
14 # Logger
15 # ~~~~~
```

**Note:** You likely need to enclose the JDBC URL configuration value with double quotes, since `:` is a reserved character in the configuration syntax.

## H2 database engine connection properties

In memory database:

```
Default database configuration using H2 database engine in an in-memory mode
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

File based database:

```
Default database configuration using H2 database engine in a persistent mode
db.default.driver=org.h2.Driver
```

```
db.default.url="jdbc:h2:/path/to/db-file"
```

The details of the H2 database URLs are found from [H2 Database Engine Cheat Sheet](#).

## SQLite database engine connection properties

```
Default database configuration using SQLite database engine
```

```
db.default.driver=org.sqlite.JDBC
```

```
db.default.url="jdbc:sqlite:/path/to/db-file"
```

## PostgreSQL database engine connection properties

```
Default database configuration using PostgreSQL database engine
```

```
db.default.driver=org.postgresql.Driver
```

```
db.default.url="jdbc:postgresql://database.example.com/playdb"
```

## MySQL database engine connection properties

```
Default database configuration using MySQL database engine
```

```
Connect to playdb as playdbuser
```

```
db.default.driver=com.mysql.jdbc.Driver
```

```
db.default.url="jdbc:mysql://localhost/playdb"
```

```
db.default.username=playdbuser
```

```
db.default.password="a strong password"
```

# How to configure several data sources

```
Orders database
```

```
db.orders.driver=org.h2.Driver
```

```
db.orders.url="jdbc:h2:mem:orders"
```

```
Customers database
```

```
db.customers.driver=org.h2.Driver
```

```
db.customers.url="jdbc:h2:mem:customers"
```

# Configuring the JDBC Driver

Play is bundled only with an [H2](#) database driver. Consequently, to deploy in production you will need to add your database driver as a dependency.

For example, if you use MySQL5, you need to add a [dependency](#) for the connector:

```
libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.34"
```

Or if the driver can't be found from repositories you can drop the driver into your project's [unmanaged dependencies](#) `lib` directory.

# Accessing the JDBC datasource

The `play.api.db` package provides access to the configured data sources:

```
import play.api.db._
```

```
val ds = DB.getDataSource()
```

# Obtaining a JDBC connection

There are several ways to retrieve a JDBC connection. The simplest way is:

```
val connection = DB.getConnection()
```

Following code show you a JDBC example very simple, working with MySQL 5.\*:

```
package controllers
import play.api.Play.current
import play.api.mvc._
import play.api.db._

object Application extends Controller {

 def index = Action {
 var outString = "Number is "
 val conn = DB.getConnection()
 try {
 val stmt = conn.createStatement
 val rs = stmt.executeQuery("SELECT 9 as testkey ")
 while (rs.next()) {
 outString += rs.getString("testkey")
 }
 } finally {
 conn.close()
 }
 Ok(outString)
 }

}
```

But of course you need to call `close()` at some point on the opened connection to return it to the connection pool. Another way is to let Play manage closing the connection for you:

```
// access "default" database
DB.withConnection { conn =>
 // do whatever you need with the connection
}
```

For a database other than the default:

```
// access "orders" database instead of "default"
DB.withConnection("orders") { conn =>
 // do whatever you need with the connection
}
```

The connection will be automatically closed at the end of the block.

**Tip:** Each `Statement` and `ResultSet` created with this connection will be closed as well.

A variant is to set the connection's auto-commit to `false` and to manage a transaction for the block:

```
DB.withTransaction { conn =>
 // do whatever you need with the connection
}
```

# Selecting and configuring the connection pool

Out of the box, Play provides two database connection pool implementations, [HikariCP](#) and [BoneCP](#). The default is HikariCP, but this can be changed by setting the `play.db.pool` property:

```
play.db.pool=bonecp
```

The full range of configuration options for connection pools can be found by inspecting the `play.db.prototype` property in Play's JDBC [reference.conf](#).

## Testing

For information on testing with databases, including how to setup in-memory databases and, see [Testing With Databases](#).

## Enabling Play database evolutions

Read [Evolutions](#) to find out what Play database evolutions are useful for, and follow the setup instructions for using it.

Next: [Using Slick to access your database](#)

# Using Play Slick

The Play Slick module makes [Slick](#) a first-class citizen of Play.

The Play Slick module consists of two features:

- Integration of Slick into Play's application lifecycle.
- Support for [Play database evolutions](#).

Play Slick currently supports Slick 3.1 with Play 2.4, for both Scala 2.10 and 2.11.

Note: This guide assumes you already know both Play 2.4 and Slick 3.1.

## Getting Help

If you are having trouble using Play Slick, check if the [FAQ](#) contains the answer. Otherwise, feel free to reach out to [play-framework user group](#). Also, note that if you are seeking help on Slick, the [slick user group](#) may be a better place.

Finally, if you prefer to get an answer for your Play and Slick questions in a timely manner, and with a well-defined SLA, you may prefer [to get in touch with Typesafe](#), as it offers commercial support for these technologies.

# About this release

If you have been using a previous version of Play Slick, you will notice that there have been quite a few major changes. It's recommended to read the [migration guide](#) for a smooth upgrade.

While, if this is the first time you are using Play Slick, you will appreciate that the integration of Slick in Play is quite austere. Meaning that if you know both Play and Slick, using Play Slick module should be straightforward.

# Setup

Add a library dependency on play-slick:

```
"com.typesafe.play" %% "play-slick" % "1.1.1"
```

The above dependency will also bring along the Slick library as a transitive dependency. This implies you don't need to add an explicit dependency on Slick, but you might still do so if needed. A likely reason for wanting to explicitly define a dependency to Slick is if you want to use a newer version than the one bundled with play-slick. Because Slick trailing dot releases are binary compatible, you won't incur any risk in using a different Slick trailing point release than the one that was used to build play-slick.

## Support for Play database evolutions

Play Slick supports [Play database evolutions](#).

To enable evolutions, you will need the following dependencies:

```
"com.typesafe.play" %% "play-slick" % "1.1.1"
"com.typesafe.play" %% "play-slick-evolutions" % "1.1.1"
```

Note there is no need to add the Play `evolutions` component to your dependencies, as it is a transitive dependency of the `play-slick-evolutions` module.

## JDBC driver dependency

Play Slick module does not bundle any JDBC driver. Hence, you will need to explicitly add the JDBC driver(s) you want to use in your application. For instance, if you would like to use an in-memory database such as H2, you will have to add a dependency to it:

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

# Database Configuration

To have Play Slick module handling the lifecycle of Slick databases, it is important that you never create database's instances explicitly in your code. Rather, you should provide a valid Slick driver and database configuration in your `application.conf` (by convention the default Slick database must be called `default`):

```
Default database configuration
slick.dbs.default.driver="slick.driver.H2Driver$"
slick.dbs.default.db.driver="org.h2.Driver"
slick.dbs.default.db.url="jdbc:h2:mem:play"
```

First, note that the above is a valid Slick configuration (for the complete list of configuration parameters that you can use to configure a database see the Slick ScalaDoc for [Database.forConfig](#) - make sure to expand the `forConfig` row in the doc).

Second, the `slick.dbs` prefix before the database's name is configurable. In fact, you may change it by overriding the value of the configuration key `play/slick/db/config`. Third, in the above configuration `slick.dbs.default.driver` is used to configure the Slick driver, while `slick.dbs.default.db.driver` is the underlying JDBC driver used by Slick's backend. In the above configuration we are configuring Slick to use H2 database, but Slick supports several other databases. Check the [Slick documentation](#) for a complete list of supported databases, and to find a matching Slick driver.

Slick does not support the `DATABASE_URL` environment variable in the same way as the default Play JDBC connection pool. But starting in version 3.0.3, Slick provides a `DatabaseUrlDataSource` specifically for parsing the environment variable.

```
slick.dbs.default.driver="slick.driver.PostgresDriver$"
slick.dbs.default.db.dataSourceClass = "slick.jdbc.DatabaseUrlDataSource"
slick.dbs.default.db.properties.driver = "org.postgresql.Driver"
```

On some platforms, such as Heroku, you may substitute the `JDBC_DATABASE_URL`, which is in the format `jdbc:vendor://host:port/db?args`, if it is available. For example:

```
slick.dbs.default.driver="slick.driver.PostgresDriver$"
slick.dbs.default.db.driver="org.postgresql.Driver"
slick.dbs.default.db.url=${JDBC_DATABASE_URL}
```

Note: Failing to provide a valid value for both `slick.dbs.default.driver` and `slick.dbs.default.db.driver` will lead to an exception when trying to run your Play application.

To configure several databases:

```
Orders database
slick.dbs.orders.driver="slick.driver.H2Driver$"
slick.dbs.orders.db.driver="org.h2.Driver"
slick.dbs.orders.db.url="jdbc:h2:mem:play"

Customers database
slick.dbs.customers.driver="slick.driver.H2Driver$"
slick.dbs.customers.db.driver="org.h2.Driver"
slick.dbs.customers.db.url="jdbc:h2:mem:play"
```

If something isn't properly configured, you will be notified in your browser:

A screenshot of a web browser window titled "Configuration error". The address bar shows "localhost:9000". The main content area has a red header with the text "Configuration error" in large white font. Below the header is a pink section containing the error message "Cannot connect to database [default]". The remainder of the page is a code editor displaying Scala code. Line 36 is highlighted in red, indicating the source of the error.

```
In /Users/mirco/Projects/oos/play-slick/samples/b
33 # You can declare as many datasources as you want
34 # By convention, the default datasource is named
35 #
36 slick.dbs.default.driver="slick.driver.OopsDriver"
37 slick.dbs.default.db.driver="org.h2.Driver"
38 slick.dbs.default.db.url="jdbc:h2:mem:play"
39
40 # Slick Evolutions
41 # ~~~~~
```

Note: Your application will be started only if you provide a valid Slick configuration.

# Usage

After having properly configured a Slick database, you can obtain a `DatabaseConfig` (which is a Slick type bundling a database and driver) in two different ways. Either by using dependency injection, or through a global lookup via the `DatabaseConfigProvider` singleton.

Note: A Slick database instance manages a thread pool and a connection pool. In general, you should not need to shut down a database explicitly in your code (by calling its `close` method), as the Play Slick module takes care of this already.

## DatabaseConfig via Dependency Injection

Here is an example of how to inject a `DatabaseConfig` instance for the default database (i.e., the database named `default` in your configuration):

```
class Application @Inject()(dbConfigProvider: DatabaseConfigProvider) extends Controller {
 val dbConfig = dbConfigProvider.get[JdbcProfile]
```

Injecting a `DatabaseConfig` instance for a different database is also easy. Simply prepend the annotation `@NamedDatabase("<db-name>")` to the `dbConfigProvider` constructor parameter:

```
class Application2 @Inject()(@NamedDatabase("<db-name>") dbConfigProvider: DatabaseConfigProvider)
extends Controller {
```

Of course, you should replace the string `"<db-name>"` with the name of the database's configuration you want to use.

For a full example, have a look at [this sample project](#).

## DatabaseConfig via Global Lookup

Here is an example of how to lookup a `DatabaseConfig` instance for the default database (i.e., the database named `default` in your configuration):

```
val dbConfig = DatabaseConfigProvider.get[JdbcProfile](Play.current)
```

Looking up a `DatabaseConfig` instance for a different database is also easy. Simply pass the database name:

```
val dbConfig = DatabaseConfigProvider.get[JdbcProfile]("<db-name>")(Play.current)
```

Of course, you should replace the string `"<db-name>"` with the name of the database's configuration you want to use.

For a full example, have a look at [this sample project](#).

## Running a database query in a Controller

To run a database query in your controller, you will need both a Slick database and driver. Fortunately, from the above we now know how to obtain a Slick `DatabaseConfig`, hence we have what we need to run a database query.

You will need to import some types and implicits from the driver:

```
import dbConfig.driver.api._
```

And then you can define a controller's method that will run a database query:

```
def index(name: String) = Action.async { implicit request =>
 val resultingUsers: Future[Seq[User]] = dbConfig.db.run(Users.filter(_.name === name).result)
 resultingUsers.map(users => Ok(views.html.index(users)))
}
```

That's just like using stock Play and Slick!

# Configuring the connection pool

Read [here](#) to find out how to configure the connection pool.

Next: [Play Slick migration guide](#)

# Play Slick Migration Guide

This is a guide for migrating from Play Slick v0.8 to v1.0 or v1.1.

It assumes you have already migrated your project to use Play 2.4 (see [Play 2.4 Migration Guide](#)), that you have read the [Slick 3.1 documentation](#), and are ready to migrate your Play application to use the new Slick Database I/O Actions API.

## Build changes

Update the Play Slick dependency in your sbt build to match the version provided in the [Setup](#) section.

### Removed H2 database dependency

Previous releases of Play Slick used to bundle the H2 database library. That's no longer the case. Hence, if you want to use H2 you will need to explicitly add it to your project's dependencies:

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

### Evolutions support in a separate module

Support for [database evolutions](#) used to be included with Play Slick. That's no longer the case. Therefore, if you are using evolutions, you now need to add an additional dependency to `play-slick-evolutions` as explained [here](#).

While, if you are not using evolutions, you can now safely remove `evolutionplugin=disabled` from your `application.conf`.

# Database configuration

With the past releases of Slick Play (which used Slick 2.1 or earlier), you used to configure Slick datasources exactly like you would configure Play JDBC datasources. This is no longer the case, and the following configuration will now be **ignored** by Play Slick:

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

There are several reasons for this change. First, the above is not a valid Slick configuration. Second, in Slick 3 you configure not just the datasource, but also both a connection pool and a thread pool. Therefore, it makes sense for Play Slick to use an entirely different path for configuring Slick databases. The default path for Slick configuration is now `slick.dbs`.

Here is how you would need to migrate the above configuration:

```
slick.dbs.default.driver="slick.driver.H2Driver$" # You must provide the required Slick driver!
slick.dbs.default.db.driver=org.h2.Driver
slick.dbs.default.db.url="jdbc:h2:mem:play"
slick.dbs.default.db.user=sa
slick.dbs.default.db.password=""
```

Note: If your database configuration contains settings for the connection pool, be aware that you will need to migrate those settings as well. However, this may be a bit trickier, because Play 2.3 default connection pool used to be BoneCP, while the default Slick 3 connection pool is HikariCP.

Read [here](#) for how to configure the connection pool.

## Automatic Slick driver detection

Play Slick used to automatically infer the needed Slick driver from the datasource configuration. This feature was removed, hence you must provide the Slick driver to use, for each Slick database configuration, in your `application.conf`.

The rationale for removing this admittedly handy feature is that we want to accept only valid Slick configurations. Furthermore, it's not always possible to automatically detect the correct Slick driver from the database configuration (if this was possible, then Slick would already provide such functionality).

Therefore, you will need to make the following changes:

- Each of your Slick database configuration must provide the Slick driver (see [here](#) for an example of how to migrate your database configuration).
- Remove all imports to `import play.api.db.slick.Config.driver.simple._`.
- Read [here](#) for how to lookup the Slick driver and database instances (which are needed to use the new Slick 3 Database I/O Actions API).

# DBAction and DBSessionRequest were removed

Play Slick used to provide a `DBAction` that was useful for:

- Conveniently pass the Slick `Session` into your Action method.
- Execute the action's body, and hence any blocking call to the database, in a separate thread pool.
- Limiting the number of blocking requests queued in the thread pool (useful to limit application's latency)

`DBAction` was indeed handy when using Slick 2.1. However, with the new Slick 3 release, we don't need it anymore. The reason is that Slick 3 comes with a new asynchronous API (a.k.a., Database I/O Actions API) that doesn't need the user to manipulate neither a `Session` nor a `Connection`. This makes `DBSessionRequest` and `DBAction`, together with its close friends `CurrentDBAction` and `PredicatedDBAction`, completely obsolete, which is why they have been removed.

Having said that, migrating your code should be as simple as changing all occurrences of `DBAction` and friends, with the standard Play `Action.async`. Click [here](#) for an example.

## Thread Pool

Play Slick used to provide a separate thread pool for executing controller's actions requiring to access a database. Slick 3 already does exactly this, hence there is no longer a need for Play Slick to create and manage additional thread pools. It follows that the below configuration parameter are effectively obsolete and should be removed from your `application.conf`:

```
db.$dbName.maxQueriesPerRequest
slick.db.execution.context
```

The parameter `db.$dbName.maxQueriesPerRequest` was used to limit the number of tasks queued in the thread pool. In Slick 3 you can reach similar results by tuning the configuration parameters `numThreads` and `queueSize`. Read the Slick ScalaDoc for [Database.forConfig](#)(make sure to expand the `forConfig` row in the doc).

While the parameter `slick.db.execution.context` was used to name the thread pools created by Play Slick. In Slick 3, each thread pool is named using the Slick database configuration path, i.e., if in your `application.conf` you have provided a Slick configuration for the database named `default`, then Slick will create a thread pool named `default` for executing the database action on the default database. Note that the name used for the thread pool is not configurable.

# Profile was removed

The trait `Profile` was removed and you can use instead `HasDatabaseConfigProvider` or `HasDatabaseConfig` with similar results.

The trait to use depend on what approach you select to retrieve a Slick database and driver (i.e., an instance of `DatabaseConfig`). If you decide to use dependency injection, then `HasDatabaseConfigProvider` will serve you well. Otherwise, use `HasDatabaseConfig`.

Read [here](#) for a discussion of how to use dependency injection vs global lookup to retrieve an instance of `DatabaseConfig`.

# Database was removed

The object `Database` was removed. To retrieve a Slick database and driver (i.e., an instance of `DatabaseConfig`) read [here](#).

# Config was removed

The `Config` object, together with `SlickConfig` and `DefaultSlickConfig`, were removed. These abstractions are simply not needed. If you used to call `Config.driver` or `Config.datasource` to retrieve the Slick driver and database, you should now use `DatabaseConfigProvider`. Read [here](#) for details.

# SlickPlayIteratees was removed

If you were using `SlickPlayIteratees.enumerateSlickQuery` to stream data from the database, you will be happy to know that doing so became a lot easier. Slick 3 implements the [reactive-streams](#) (Service Provider Interface), and Play 2.4 provides a utility class to handily convert a reactive stream into a Play enumerator.

In Slick, you can obtain a reactive stream by calling the method `stream` on a Slick database instance (instead of the eager `run`). To convert the stream into an enumerator simply call `play.api.libs.streams.Streams.publisherToEnumerator`, passing the stream in argument.

For a full example, have a look at [this sample project](#).

# DDL support was removed

Previous versions of Play Slick included a DDL plugin which would read your Slick tables definitions, and automatically creates schema updates on reload. While this is an interesting and useful feature, the underlying implementation was fragile, and relied on the assumption that your tables would be accessible via a module (i.e., a Scala `object`). This coding pattern was possible because Play Slick allowed to import the Slick driver available via a top-level import. However, because support for [automatic detection of the Slick driver](#) was removed, you will not declare a top-level import for the Slick driver. This implies that Slick tables will no longer be accessible via a module. This fact breaks the assumption made in the initial implementation of the DDL plugin, and it's the reason why the feature was removed.

The consequence of the above is that you are in charge of creating and managing your project's database schema. Therefore, whenever you make a change a Slick table in the code, make sure to also update the database schema. If you find it tedious to manually keep in sync your database schema and the related table definition in your code, you may want to have a look at the code generation feature available in Slick.

Next: [Play Slick advanced topics](#)

# Play Slick Advanced Topics

## Connection Pool

With Slick 3 release, Slick starts and controls both a connection pool and a thread pool for optimal asynchronous execution of your database actions.

In Play Slick we have decided to let Slick be in control of creating and managing the connection pool (the default connection pool used by Slick 3 is [HikariCP](#)), which means that to tune the connection pool you will need to look at the Slick ScalaDoc [forDatabase.forConfig](#)(make sure to expand the `forConfig` row in the doc). In fact, be aware that any value you may pass for setting the Play connection pool (e.g., under the key `play.db.default.hikaricp`) is simply not picked up by Slick, and hence effectively ignored.

Also, note that as stated in the [Slick documentation](#), a reasonable default for the connection pool size is calculated from the thread pool size. In fact, you should only need to tune `numThreads` and `queueSize` in most cases, for each of your database configuration. Finally, it's worth mentioning that while Slick allows using a different connection pool than [HikariCP](#)(though, Slick currently only offers built-in support for HikariCP, and requires

you to provide an implementation of `JdbcDataSourceFactory` if you want to use a different connection pool), Play Slick currently doesn't allow using a different connection pool than HikariCP.

Note: Changing the value of `play.db.pool` won't affect what connection pool Slick is using. Furthermore, be aware that any configuration under `play.db` is not considered by Play Slick.

## Thread Pool

With Slick 3.0 release, Slick starts and controls both a thread pool and a connection pool for optimal asynchronous execution of your database actions.

For optimal execution, you may need to tune the `numThreads` and `queueSize` parameters, for each of your database configuration. Refer to the [Slick documentation](#) for details.

Next: [Play Slick FAQ](#)

## Play Slick FAQ

### What version should I use?

Have a look at the [compatibility matrix](#) to know what version you should be using.

### `play.db.pool` is ignored

It's indeed the case. Changing the value of `play.db.pool` won't affect what connection pool Slick is going to use. The reason is simply that Play Slick module currently doesn't support using a different connection pool than [HikariCP](#).

### Changing the connection pool used by Slick

While Slick allows using a different connection pool than [HikariCP](#)(though, Slick currently only offers built-in support for HikariCP, and requires you to provide an implementation of `JdbcDataSourceFactory` if you want to use a different connection pool), Play Slick currently doesn't allow using a different connection pool than HikariCP. If you find yourself needing this feature, you can try to drop us a note on [playframework-dev](#).

# A binding to play.api.db.DBApi was already configured

If you get the following exception when starting your Play application:

1) A binding to play.api.db.DBApi was already configured at play.api.db.slick.evolutions.EvolutionsModule.bindings:  
Binding(interface play.api.db.DBApi to ConstructionTarget(class play.api.db.slick.evolutions.internal.DBAdapter) in interface javax.inject.Singleton).  
at play.api.db.DBModule.bindings(DBModule.scala:25):  
Binding(interface play.api.db.DBApi to ProviderConstructionTarget(class play.api.db.DBApiProvider))  
It is very likely that you have enabled the jdbc plugin, and that doesn't really make sense if you are using Slick for accessing your databases. To fix the issue simply remove the Play *jdbc* component from your project's build.  
Another possibility is that there is another Play module that is binding DBApi to some other concrete implementation. This means that you are still trying to use Play Slick together with another Play module for database access, which is likely not what you want.

# Play throws java.lang.ClassNotFoundException: org.h2.tools.Server

If you get the following exception when starting your Play application:

```
java.lang.ClassNotFoundException: org.h2.tools.Server
 at java.net.URLClassLoader$1.run(URLClassLoader.java:372)
 at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
 ...
...
```

It means you are trying to use a H2 database, but have forgot to add a dependency to it in your project's build. Fixing the problem is simple, just add the missing dependency in your project's build, e.g.,

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

Next: Using Anorm to access your database

# Anorm, simple SQL data access

Play includes a simple data access layer called Anorm that uses plain SQL to interact with the database and provides an API to parse and transform the resulting datasets.

## Anorm is Not an 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 instructions, and configure it as explained [on the Scala database page](#) .

## Overview

It can feel strange to return to plain old SQL to access an SQL database these days, especially for Java developers accustomed to using a high-level Object Relational Mapper like Hibernate to completely hide this aspect.

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

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

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

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, the goal of the Play Scala SQL access layer is to provide several APIs to effectively transform JDBC data into other Scala structures.

## *You don't need another DSL to access relational databases*

SQL is already the best DSL for accessing 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 by not using a particular database's interesting features.

Play will sometimes provide you with pre-filled SQL statements, but the idea is not to hide the fact that we use SQL under the hood. Play just saves typing a bunch of characters for trivial queries, and you can always fall back 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 checks your queries based on a meta-model definition that you often write yourself by 'mapping' your data structure to the database schema.

There are no guarantees that this meta-model is correct. Even if the compiler says that your 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 Relational Mapping works well for trivial cases, but when you have to deal with complex schemas or existing databases, you will spend most of your time fighting with your ORM to make it generate the SQL queries you want.

Writing SQL queries yourself 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 full control of your SQL code.

---

# Add Anorm to your project

You will need to add Anorm and JDBC plugin to your dependencies :

```
libraryDependencies ++= Seq(
 jdbc,
 "com.typesafe.play" %% "anorm" % "2.4.0"
)
```

# Executing SQL queries

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

First, import `anorm._`, and then simply use the `SQL` object to create queries. You need a `Connection` to run a query, and you can retrieve one from the `play.api.db.DB` helper:

```
import anorm._
import play.api.db.DB
```

```
DB.withConnection { implicit c =>
 val result: Boolean = SQL("Select 1").execute()
}
```

The `execute()` method returns a Boolean value indicating whether the execution was successful.

To execute an update, you can use `executeUpdate()`, which returns the number of rows updated.

```
val result: Int = SQL("delete from City where id = 99").executeUpdate()
```

If you are inserting data that has an auto-generated `Long` primary key, you can call `executeInsert()`.

```
val id: Option[Long] =
 SQL("insert into City(name, country) values ({name}, {country})")
 .on('name -> "Cambridge", 'country -> "New Zealand").executeInsert()
```

When key generated on insertion is not a single `Long`, `executeInsert` can be passed a `ResultSetParser` to return the correct key.

```
import anorm.SqlParser.str
```

```
val id: List[String] =
 SQL("insert into City(name, country) values ({name}, {country})")
 .on('name -> "Cambridge", 'country -> "New Zealand")
 .executeInsert(str.+) // insertion returns a list of at least one string keys
```

Since Scala supports multi-line strings, feel free to use them for complex SQL statements:

```
val 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 later assign a value to them:

```
SQL(
 """
 select * from Country c
 """)
```

```

join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {countryCode};
"""
).on("countryCode" -> "FRA")

```

You can also use string interpolation to pass parameters (see details thereafter).

In case several columns are found with same name in query result, for example columns named `code` in both `Country` and `CountryLanguage` tables, there can be ambiguity. By default a mapping like following one will use the last column:

```
import anorm.{ SQL, SqlParser }
```

```

val code: String = SQL(
"""
 select * from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {countryCode}
"""
).on("countryCode" -> "FRA").as(SqlParser.str("code").single)

```

If `Country.Code` is 'First' and `CountryLanguage` is 'Second', then in previous example `code` value will be 'Second'. Ambiguity can be resolved using qualified column name, with table name:

```
import anorm.{ SQL, SqlParser }
```

```

val code: String = SQL(
"""
 select * from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {countryCode}
"""
).on("countryCode" -> "FRA").as(SqlParser.str("Country.code").single)
// code == "First"

```

When a column is aliased, typically using SQL `AS`, its value can also be resolved. Following example parses column with `country_lang` alias.

```
import anorm.{ SQL, SqlParser }
```

```

val lang: String = SQL(
"""
 select l.language AS country_lang from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {countryCode}
"""
).on("countryCode" -> "FRA").
 as(SqlParser.str("country_lang").single)

```

Columns can also be specified by position, rather than name:

```

import anorm.SqlParser.{ str, float }
// Parsing column by name or position
val parser =
 str("name") ~ float(3) /* third column as float */ map {

```

```

case name ~ f => (name -> f)
}

val product: (String, Float) = SQL("SELECT * FROM prod WHERE id = {id}").
 on('id -> "p").as(parser.single)

```

`java.util.UUID` can be used as parameter, in which case its string value is passed to statement.

## SQL queries using String Interpolation

Since Scala 2.10 supports custom String Interpolation there is also a 1-step alternative to `SQL(queryString).on(params)` seen before. You can abbreviate the code as:

```

val name = "Cambridge"
val country = "New Zealand"

```

```
SQL"insert into City(name, country) values ($name, $country)"
```

It also supports multi-line string and inline expresions:

```

val lang = "French"
val population = 10000000
val margin = 500000

val code: String = SQL"""
 select * from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where l.Language = $lang and c.Population >= ${population - margin}
 order by c.Population desc limit 1"""
 .as(SqlParser.str("Country.code")).single

```

This feature tries to make faster, more concise and easier to read the way to retrieve data in Anorm. Please, feel free to use it wherever you see a combination of `SQL().on()` functions (or even an only `SQL()` without parameters).

By using `#$value` instead of `$value`, interpolated value will be part of the prepared statement, rather being passed as a parameter when executing this SQL statement (e.g. `#$cmd` and `#$table` in example bellow).

```

val cmd = "SELECT"
val table = "Test"

SQL"""$cmd * FROM #$table WHERE id = ${"id1"} AND code IN (${Seq(2, 5)})"""

// prepare the SQL statement, with 1 string and 2 integer parameters:
// SELECT * FROM Test WHERE id = ? AND code IN (?, ?)

```

# Streaming results

Query results can be processed row per row, not having all loaded in memory.

In the following example we will count the number of country rows.

```
val countryCount: Either[List[Throwable], Long] =
 SQL"Select count(*) as c from Country".fold(0L) { (c, _) => c + 1 }
In previous example, either it's the successful Long result (right), or the list of errors (left).
```

Result can also be partially processed:

```
val books: Either[List[Throwable], List[String]] =
 SQL("Select name from Books").foldWhile(List[String]()) { (list, row) =>
 if (list.size == 100) (list -> false) // stop with `list`
 else (list ::= row[String]("name")) -> true // continue with one more name
 }
```

It's possible to use a custom streaming:

```
import anorm.{ Cursor, Row }

@annotation.tailrec
def go(c: Option[Cursor], l: List[String]): List[String] = c match {
 case Some(cursor) => {
 if (l.size == 100) l // custom limit, partial processing
 else {
 go(cursor.next, l :+ cursor.row[String]("name"))
 }
 }
 case _ => l
}
```

```
val books: Either[List[Throwable], List[String]] =
 SQL("Select name from Books").withResult(go(_, List.empty[String]))
```

The parsing API can be used with streaming, using `RowParser` on each cursor `.row`. The previous example can be updated with row parser.

```
import scala.util.{ Try, Success => TrySuccess, Failure }
```

```
// bookParser: anorm.RowParser[Book]
```

```
@annotation.tailrec
def go(c: Option[Cursor], l: List[Book]): Try[List[Book]] = c match {
 case Some(cursor) => {
 if (l.size == 100) l // custom limit, partial processing
 else {
 val parsed: Try[Book] = cursor.row.as(bookParser)

 parsed match {
 case TrySuccess(book) => // book successfully parsed from row
 go(cursor.next, l :+ book)
 case Failure(f) => /* fails to parse a book */ Failure(f)
 }
 }
 }
 case _ => l
}
```

```

val books: Either[List[Throwable], Try[List[Book]]] =
 SQL("Select name from Books").withResult(go(_, List.empty[Book]))

books match {
 case Left(streamingErrors) => ???
 case Right(Failure(parsingError)) => ???
 case Right(TrySuccess(listOfBooks)) => ???
}

```

## Multi-value support

Anorm parameter can be multi-value, like a sequence of string.  
In such case, values will be prepared to be passed to JDBC.

```

// With default formatting (", " as separator)
SQL("SELECT * FROM Test WHERE cat IN ({categories})").
 on('categories -> Seq("a", "b", "c")
// -> SELECT * FROM Test WHERE cat IN ('a', 'b', 'c')

// With custom formatting
import anorm.SeqParameter
SQL("SELECT * FROM Test t WHERE {categories}").
 on('categories -> SeqParameter(
 values = Seq("a", "b", "c"), separator = " OR ",
 pre = "EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name=",
 post = ")")
/* ->
SELECT * FROM Test t WHERE
EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='a')
OR EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='b')
OR EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='c')
*/

```

On purpose multi-value parameter must strictly be declared with one of supported types (`List`, `'Seq`, `Set`, `SortedSet`, `Stream`, `Vector` and `SeqParameter`). Value of a subtype must be passed as parameter with supported:

```

val seq = IndexedSeq("a", "b", "c")
// seq is instance of Seq with inferred type IndexedSeq[String]

```

```

// Wrong
SQL"SELECT * FROM Test WHERE cat in ($seq)"
// Erroneous - No parameter conversion for IndexedSeq[T]

```

```

// Right
SQL"SELECT * FROM Test WHERE cat in (${seq: Seq[String]})"

```

```

// Right
val param: Seq[String] = seq
SQL"SELECT * FROM Test WHERE cat in ($param)"

```

In case parameter type is JDBC array (`java.sql.Array`), its value can be passed as `Array[T]`, as long as element type `T` is a supported one.

```

val arr = Array("fr", "en", "ja")
SQL"UPDATE Test SET langs = $arr".execute()

```

A column can also be multi-value if its type is JDBC array (`java.sql.Array`), then it can be mapped to either array or list (`Array[T]` or `List[T]`), provided type of element (`T`) is also supported in column mapping.

```
import anorm.SQL
import anorm.SqlParser.{ scalar, * }

// array and element parser
import anorm.Column.{ columnToArray, stringToArray }
```

```
val res: List[Array[String]] =
 SQL("SELECT str_arr FROM tbl").as(scalar[Array[String]].*)
```

Convenient parsing functions is also provided for arrays with `SqlParser.array[T](...)` and `SqlParser.list[T](...)`.

## Batch update

When you need to execute SQL statement several times with different arguments, batch query can be used (e.g. to execute a batch of insertions).

```
import anorm.BatchSql

val batch = BatchSql(
 "INSERT INTO books(title, author) VALUES({title}, {author})",
 Seq[NamedParameter]("title" -> "Play 2 for Scala", "author" -> "Peter Hilton"),
 Seq[NamedParameter]("title" -> "Learning Play! Framework 2",
 "author" -> "Andy Petrella"))
```

```
val res: Array[Int] = batch.execute() // array of update count
```

Batch update must be called with at least one list of parameter. If a batch is executed with the mandatory first list of parameter being empty (e.g. `Nil`), only one statement will be executed (without parameter), which is equivalent to `SQL(statement).executeUpdate()`.

## Edge cases

Passing anything different from string or symbol as parameter name is now deprecated. For backward compatibility, you can

```
activate anorm.features.parameterWithUntypedName.
```

```
import anorm.features.parameterWithUntypedName // activate
```

```
val untyped: Any = "name" // deprecated
```

```
SQL("SELECT * FROM Country WHERE {p}").on(untyped -> "val")
```

Type of parameter value should be visible, to be properly set on SQL statement.

Using value as `Any`, explicitly or due to erasure, leads to compilation error `No implicit view available from Any => anorm.ParameterValue`.

```
// Wrong #1
```

```
val p: Any = "strAsAny"
```

```
SQL("SELECT * FROM test WHERE id={id}").
```

```
on(Id -> p) // Erroneous - No conversion Any => ParameterValue
```

```
// Right #1
```

```
val p = "strAsString"
```

```

SQL("SELECT * FROM test WHERE id={id}").on('id -> p)

// Wrong #2
val ps = Seq("a", "b", 3) // inferred as Seq[Any]
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
 on('a -> ps(0), // ps(0) - No conversion Any => ParameterValue
 'b -> ps(1),
 'c -> ps(2))

// Right #2
val ps = Seq[anorm.ParameterValue]("a", "b", 3) // Seq[ParameterValue]
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
 on('a -> ps(0), 'b -> ps(1), 'c -> ps(2))

// Wrong #3
val ts = Seq(// Seq[(String -> Any)] due to _2
 "a" -> "1", "b" -> "2", "c" -> 3)

val nps: Seq[NamedParameter] = ts map { t =>
 val p: NamedParameter = t; p
 // Erroneous - no conversion (String,Any) => NamedParameter
}

SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").on(nps :_*)

```

```

// Right #3
val nps = Seq[NamedParameter](// Tuples as NamedParameter before Any
 "a" -> "1", "b" -> "2", "c" -> 3)
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
 on(nps: _*) // Fail - no conversion (String,Any) => NamedParameter

```

For backward compatibility, you can activate such unsafe parameter conversion, accepting untyped `Any` value, with `anorm.features.anyToStatement`.

```
import anorm.features.anyToStatement
```

```

val d = new java.util.Date()
val params: Seq[NamedParameter] = Seq("mod" -> d, "id" -> "idv")
// Values as Any as heterogenous

```

```
SQL("UPDATE item SET last_modified = {mod} WHERE id = {id}").on(params:_*)
```

It's not recommended because moreover hiding implicit resolution issues, as untyped it could lead to runtime conversion error, with values are passed on statement using `setObject`.

In previous example, `java.util.Date` is accepted as parameter but would with most databases raise error (as it's not valid JDBC type).

In some cases, some JDBC drivers returns a result set positioned on the first row rather than `before this first row` (e.g. stored procedure with Oracle JDBC driver).

To handle such edge-case, `.withResultSetOnFirstRow(true)` can be used as following.

```
SQL("EXEC stored_proc {arg}").on("arg" -> "val").withResultSetOnFirstRow(true)
SQL"""\EXEC stored_proc ${"val"}""".withResultSetOnFirstRow(true)
```

# 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 transforms 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 WHERE id = {i}").
 on("i" -> "id").map({
 case Row("France", _) => France()
 case Row(name:String, pop:Int) if(pop > 1000000) => BigCountry(name)
 case Row(name:String, _) => SmallCountry(name)
 }).list
```

# Using for-comprehension

Row parser can be defined as for-comprehension, working with SQL result type. It can be useful when working with lot of column, possibly to work around case class limit.

```
import anorm.SqlParser.{ str, int }

val parser = for {
 a <- str("colA")
 b <- int("colB")
} yield (a -> b)

val parsed: (String, Int) = SELECT("SELECT * FROM Test").as(parser.single)
```

# Retrieving data along with execution context

Moreover data, query execution involves context information like SQL warnings that may be raised (and may be fatal or not), especially when working with stored SQL procedure.

Way to get context information along with query data is to use `executeQuery()`:

```
import anorm.SqlQueryResult
```

```
val res: SqlQueryResult = SQL("EXEC stored_proc {code}").
```

```

on('code -> code).executeQuery()

// Check execution context (there warnings) before going on
val str: Option[String] =
 res.createStatementWarning match {
 case Some(warning) =>
 warning.printStackTrace()
 None
 case _ => res.as(scalar[String].singleOpt) // go on row parsing
 }
}

```

# Working with optional/nullable values

If a column in database can contain `Null` values, you need to parse it as an `Option` type. For example, the `indepYear` of the `Country` table is nullable, so you need to match it `asOption[Int]`:

```

case class Info(name: String, year: Option[Int])

val parser = str("name") ~ get[Option[Int]]("indepYear") map {
 case n ~ y => Info(n, y)
}

```

```
val res: List[Info] = SQL("Select name,indepYear from Country").as(parser.*)
```

If you try to match this column as `Int` it won't be able to parse `Null` values. Suppose you try to retrieve the column content as `Int` directly from the dictionary:

```

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

```

This will produce an `UnexpectedNullableFound(COUNTRY.INDEPYEAR)` exception if it encounters a null value, so you need to map it properly to an `Option[Int]`.

A nullable parameter is also passed as `Option[T]`, `T` being parameter base type (see `Parameters` section thereafter).

Passing directly `None` for a NULL value is not supported, as inferred as `Option[Nothing]` (`Nothing` being unsafe for a parameter value). In this case, `Option.empty[T]` must be used.

`// OK:`

```
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> Some("Title"))
```

```
val title1 = Some("Title1")
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> title1)
```

```
val title2: Option[String] = None
// None inferred as Option[String] on assignment
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> title2)
```

```
// Not OK:
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> None)

// OK:
SQL("INSERT INTO Test(title) VALUES(${Option.empty[String]})")
```

# Using the Parser API

You can use the parser API to create generic and reusable parsers that can parse the result of any select query.

**Note:** This is really useful, since most queries in a web application will return similar data sets. For example, if you have defined a parser able to parse a `Country` from a result set, and another `Language` parser, you can then easily compose them to parse both `Country` and `Language` from a join query.

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

## Getting a single result

First you need a `RowParser`, i.e. a parser able to parse one row to a Scala value. For example we can define a parser to transform a single column result set row, to a `ScalaLong`:

```
val rowParser = scalar[Long]
```

Then we have to transform it into a `ResultSetParser`. Here we will create a parser that parse a single row:

```
val rsParser = scalar[Long].single
```

So this parser will parse a result set to return a `Long`. It is useful to parse to result produced by a simple SQL `select count` query:

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

If expected single result is optional (0 or 1 row), then `scalar` parser can be combined with `singleOpt`:

```
val name: Option[String] =
 SQL("SELECT name FROM Country WHERE code = $code" as scalar[String].singleOpt)
```

## Getting a single optional result

Let's say you want to retrieve the `country_id` from the country name, but the query might return null. We'll use the `singleOpt` parser :

```
val countryId: Option[Long] =
 SQL("SELECT country_id FROM Country C WHERE C.country='France'")
 .as(scalar[Long].singleOpt)
```

## Getting a more complex result

Let's write a more complicated parser:

`str("name") ~ int("population")`, will create a `RowParser` able to parse a row containing a String `name` column and an Integer `population` column. Then we can create a `ResultSetParser` that will parse as many rows of this kind as it can, using `*`:

```
val populations: List[String ~ Int] =
 SQL("SELECT * FROM Country").as((str("name") ~ int("population")).*)
```

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

You can also rewrite the same code as:

```
val result: List[String ~ Int] = SQL("SELECT * FROM Country").
 as((get[String]("name") ~ get[Int]("population")).*)
```

Now what about the `String~Int` type? This is an **Anorm** type that is not really convenient to use outside of your database access code. You would rather have a simple tuple `(String, Int)` instead. You can use the `map` function on a `RowParser` to transform its result to a more convenient type:

```
val parser = str("name") ~ int("population") map { case n ~ p => (n, p) }
```

**Note:** We created a tuple `(String, Int)` here, but there is nothing stopping you from transforming the `RowParser` result to any other type, such as a custom case class.

Now, because transforming `A ~ B ~ C` types to `(A, B, C)` is a common task, we provide a `flatten` function that does exactly that. So you finally write:

```
val result: List[(String, Int)] =
 SQL("select * from Country").as(parser.flatten.*)
```

A `RowParser` can be combined with any function to apply it with extracted columns.

```
import anorm.SqlParser.{ int, str, to }
```

```
def display(name: String, population: Int): String =
 s"The population in $name is of $population."
```

```
val parser = str("name") ~ int("population") map (to(display _))
```

**Note:** The mapping function must be partially applied (syntax `fn _`) when given `to` the parser (see [SLS 6.26.2, 6.26.5 - Eta expansion](#)).

If list should not be empty, `parser.+` can be used instead of `parser.*`.

Anorm is providing parser combinators other than the most common `~` one: `~>`, `<~`.

```
import anorm.{ SQL, SqlParser }, SqlParser.{ int, str }
```

```
// Combinator ~>
```

```
val String = SQL("SELECT * FROM test").as((int("id") ~> str("val")).single)
// row has to have an int column 'id' and a string 'val' one,
// keeping only 'val' in result
```

```
val Int = SQL("SELECT * FROM test").as((int("id") <~ str("val")).single)
// row has to have an int column 'id' and a string 'val' one,
// keeping only 'id' in result
```

## A more complicated example

Now let's try with a more complicated example. How to parse the result of the following query to retrieve the country name and all spoken languages for a country code?

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

Let's start by parsing all rows as a `List[(String, String)]` (a list of name,language tuple):

```
var p: ResultSetParser[List[(String, String)]] = {
 str("name") ~ str("language") map(flatten) *
}
```

Now we get this kind of result:

```
List(
 ("France", "Arabic"),
 ("France", "French"),
 ("France", "Italian"),
 ("France", "Portuguese"),
 ("France", "Spanish"),
 ("France", "Turkish")
)
```

We can then use the Scala collection API, to transform it to the expected result:

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

languages.headOption.map { f =>
 SpokenLanguages(f._1, languages.map(_._2))
}
```

Finally, we get this convenient function:

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

def spokenLanguages(countryCode: String): Option[SpokenLanguages] = {
 val languages: List[(String, String)] = SQL(
 """
 select c.name, l.language from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {code};
 """
)
 .on("code" -> countryCode)
 .as(str("name") ~ str("language") map(flatten) *)

 languages.headOption.map { f =>
 SpokenLanguages(f._1, languages.map(_._2))
 }
}
```

```
}
```

To continue, let's complicate our example to separate the official language from the others:

```
case class SpokenLanguages(
 country:String,
 officialLanguage: Option[String],
 otherLanguages:Seq[String]
)

def spokenLanguages(countryCode: String): Option[SpokenLanguages] = {
 val languages: List[(String, String, Boolean)] = SQL(
 """
 select * from Country c
 join CountryLanguage l on l.CountryCode = c.Code
 where c.code = {code};
 """
).on("code" -> countryCode)
 .as {
 str("name") ~ str("language") ~ str("isOfficial") map {
 case n~l~"T" => (n,l,true)
 case n~l~"F" => (n,l,false)
 } *
 }
 languages.headOption.map { f =>
 SpokenLanguages(
 f._1,
 languages.find(f._3).map(f._2),
 languages.filterNot(f._3).map(f._2)
)
 }
}
```

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

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

## JDBC mappings

As already seen in this documentation, Anorm provides builtins converters between JDBC and JVM types.

## Column parsers

Following table describes which JDBC numeric types (getters on `java.sql.ResultSet`, first column) can be parsed to which Java/Scala types (e.g. integer column can be read as double value).

↓JDBC / JVM→	BigDecimal <sup>1</sup>	BigInteger <sup>2</sup>	Boolean	Byte	Double	Float	Int	Long	Short
BigDecimal <sup>1</sup>	Yes	Yes	No	No	Yes	No	Yes	Yes	No
BigInteger <sup>2</sup>	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No
Boolean	No	No	Yes	Yes	No	No	Yes	Yes	Yes
Byte	Yes	No	No	Yes	Yes	Yes	No	No	Yes
Double	Yes	No	No	No	Yes	No	No	No	No
Float	Yes	No	No	No	Yes	Yes	No	No	No
Int	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No
Long	Yes	Yes	No	No	No	No	Yes	Yes	No
Short	Yes	No	No	Yes	Yes	Yes	No	No	Yes

- 1. Types `java.math.BigDecimal` and `scala.math.BigDecimal`.
- 1. Types `java.math.BigInteger` and `scala.math.BigInt`.

The second table shows mappings for the other supported types.

↓JDBC / JVM→	Array[T] <sup>3</sup>	Char	List <sup>3</sup>	String	UUID <sup>4</sup>
Array <sup>5</sup>	Yes	No	Yes	No	No
Clob	No	Yes	No	Yes	No
Iterable <sup>6</sup>	Yes	No	Yes	No	No

↓ JDBC / JVM →	Array[T] <sup>3</sup>	Char	List <sup>3</sup>	String	UUID <sup>4</sup>
Long	No	No	No	No	No
String	No	Yes	No	Yes	Yes
UUID	No	No	No	No	Yes

- 1. Array which type `T` of elements is supported.
- 1. Type `java.util.UUID`.
- 1. Type `java.sql.Array`.
- 1. Type `java.lang.Iterable[_]`.  
Optional column can be parsed as `Option[T]`, as soon as `T` is supported.

Binary data types are also supported.

↓ JDBC / JVM →	Array[Byte]	InputStream <sup>1</sup>
Array[Byte]	Yes	Yes
Blob <sup>2</sup>	Yes	Yes
Clob <sup>3</sup>	No	No
InputStream <sup>4</sup>	Yes	Yes
Reader <sup>5</sup>	No	No

- 1. Type `java.io.InputStream`.
- 1. Type `java.sql.Blob`.
- 1. Type `java.sql.Clob`.
- 1. Type `java.io.Reader`.

CLOBs/TEXTs can be extracted as so:

```
SQL("Select name,summery from Country")().map {
 case Row(name: String, summary: java.sql.Clob) => name -> summary
}
```

Here we specifically chose to use `map`, as we want an exception if the row isn't in the format we expect.

Extracting binary data is similarly possible:

```
SQL("Select name,image from Country")().map {
 case Row(name: String, image: Array[Byte]) => name -> image
}
```

For types where column support is provided by Anorm, convenient functions are available to ease writing custom parsers. Each of these functions parses column either by name or index (> 1).

Type	Function
Array[Byte]	byteArray
Boolean	bool
Byte	byte
Date	date
Double	double
Float	float
InputStream <sup>1</sup>	binaryStream
Int	int
Long	long
Short	short

Type	Function			
String	str			
• 1. Type <code>java.io.InputStream</code> .				
The <a href="#">Joda</a> and <a href="#">Java 8</a> temporal types are also supported. ↓ JDBC / JVM →	Date <sup>1</sup>	DateTime <sup>2</sup>	Instant <sup>3</sup>	
Date	Yes	Yes	Yes	
Long	Yes	Yes	Yes	
Timestamp	Yes	Yes	Yes	
Timestamp wrapper <sup>5</sup>	Yes	Yes	Yes	
• 1. Type <code>java.util.Date</code> .				
• 1. Types <code>org.joda.time.DateTime</code> , <code>java.time.LocalDateTime</code> and <code>java.time.ZonedDateTime</code> .				
• 1. Type <code>org.joda.time.Instant</code> and <code>java.time.Instant</code> (see Java 8).				
• 1. Any type with a getter <code>getTimestamp</code> returning a <code>java.sql.Timestamp</code> . It's possible to add custom mapping, for example if underlying DB doesn't support boolean datatype and returns integer instead. To do so, you have to provide a new implicit conversion for <code>Column[T]</code> , where <code>T</code> is the target Scala type: <code>import anorm.Column</code>				
<pre>// Custom conversion from JDBC column to Boolean implicit def columnToBoolean: Column[Boolean] =   Column.nonNull1 { (value, meta) =&gt;     val MetaDataItem(qualified, nullable, clazz) = meta     value match {       case bool: Boolean =&gt; Right(bool) // Provided-default case       case bit: Int     =&gt; Right(bit == 1) // Custom conversion       case _              =&gt; Left(TypeDoesNotMatch(s"Cannot convert \$value:         \${value.asInstanceOf[AnyRef].getClass} to Boolean for column \$qualified"))     }   }</pre>				

## Parameters

The following table indicates how JVM types are mapped to JDBC parameter types:

JVM	JDBC	Nullable
Array[T] <sup>1</sup>	Array <sup>2</sup> with $\boxed{T}$ mapping for each element	Yes
BigDecimal <sup>3</sup>	BigDecimal	Yes
BigInteger <sup>4</sup>	BigDecimal	Yes
Boolean <sup>5</sup>	Boolean	Yes
Byte <sup>6</sup>	Byte	Yes
Char <sup>7</sup> /String	String	Yes
Date/Timestamp	Timestamp	Yes
Double <sup>8</sup>	Double	Yes
Float <sup>9</sup>	Float	Yes
Int <sup>10</sup>	Int	Yes
List[T]	Multi-value <sup>11</sup> , with $\boxed{T}$ mapping for each element	No
Long <sup>12</sup>	Long	Yes
Object <sup>13</sup>	Object	Yes
Option[T]	$\boxed{T}$ being type if some defined value	No
Seq[T]	Multi-value, with $\boxed{T}$ mapping for each element	No
Set[T] <sup>14</sup>	Multi-value, with $\boxed{T}$ mapping for each element	No
Short <sup>15</sup>	Short	Yes

JVM	JDBC	Nullable
SortedSet[T] <sup>16</sup>	Multi-value, with <code>T</code> mapping for each element	No
Stream[T]	Multi-value, with <code>T</code> mapping for each element	No
UUID	String <sup>17</sup>	No
Vector	Multi-value, with <code>T</code> mapping for each element	No
•	1. Type Scala <code>Array[T]</code> .	
•	1. Type <code>java.sql.Array</code> .	
•	1. Types <code>java.math.BigDecimal</code> and <code>scala.math.BigDecimal</code> .	
•	1. Types <code>java.math.BigInteger</code> and <code>scala.math.BigInt</code> .	
•	1. Types <code>Boolean</code> and <code>java.lang.Boolean</code> .	
•	1. Types <code>Byte</code> and <code>java.lang.Byte</code> .	
•	1. Types <code>Char</code> and <code>java.lang.Character</code> .	
•	1. Types compatible with <code>java.util.Date</code> , and any wrapper type with <code>getTimestamp</code> : <code>java.sql.Timestamp</code> .	
•	1. Types <code>Double</code> and <code>java.lang.Double</code> .	
•	1. Types <code>Float</code> and <code>java.lang.Float</code> .	
•	1. Types <code>Int</code> and <code>java.lang.Integer</code> .	
•	1. Types <code>Long</code> and <code>java.lang.Long</code> .	
•	1. Type <code>anorm.Object</code> , wrapping opaque object.	
•	1. Multi-value parameter, with one JDBC placeholder ( <code>?</code> ) added for each element.	
•	1. Type <code>scala.collection.immutable.Set</code> .	
•	1. Types <code>Short</code> and <code>java.lang.Short</code> .	

- 1. Type `scala.collection.immutable.SortedSet`.
  - 1. Not-null value extracted using `.toString`.  
Passing `None` for a nullable parameter is deprecated, and typesafe `Option.empty[T]` must be used instead.
- Large and stream parameters are also supported.
- | JVM                      | JDBC           |
|--------------------------|----------------|
| Array[Byte]              | Long varbinary |
| Blob <sup>1</sup>        | Blob           |
| InputStream <sup>2</sup> | Long varbinary |
| Reader <sup>3</sup>      | Long varchar   |
- 1. Type `java.sql.Blob`
  - 1. Type `java.io.InputStream`
  - 1. Type `java.io.Reader`  
Joda and Java 8 temporal types are supported as parameters:
- | JVM                        | JDBC      |
|----------------------------|-----------|
| DateTime <sup>1</sup>      | Timestamp |
| Instant <sup>2</sup>       | Timestamp |
| LocalDateTime <sup>3</sup> | Timestamp |
| ZonedDateTime <sup>4</sup> | Timestamp |
- 1. Type `org.joda.time.DateTime`.
  - 1. Type `org.joda.time.Instant` and `java.time.Instant`.
  - 1. Type `org.joda.time.LocalDateTime`.

1. Type `org.joda.time.ZonedDateTime`

Custom or specific DB conversion for parameter can also be provided:

```
import java.sql.PreparedStatement
import anorm.ToStatement

// Custom conversion to statement for type T
implicit def customToStatement: ToStatement[T] = new ToStatement[T] {
 def set(statement: PreparedStatement, i: Int, value: T): Unit =
 ??? // Sets |value| on |statement|
}
```

If involved type accept `null` value, it must be appropriately handled in conversion.

The `NotNullGuard` trait can be used to explicitly refuse `null` values in parameter conversion: `new ToStatement[T] with NotNullGuard { /* ... */ }`.

DB specific parameter can be explicitly passed as opaque value.

In this case at your own risk, `setObject` will be used on statement.

```
val anyVal: Any = myVal
SQL("UPDATE t SET v = {opaque}").on('opaque -> anorm.Object(anyVal))
```

Next: [Integrating with other database access libraries](#)

# Integrating with other database libraries

You can use any `SQL` database access library you like with Play, and easily retrieve either a `Connection` or a `Datasource` from the `play.api.db.DB` helper.

# Integrating with ScalaQuery

From here you can integrate any JDBC access layer that needs a JDBC data source. For example, to integrate with `ScalaQuery`:

```
import play.api.db._
import play.api.Play.current

import org.scalaquery.ql._
import org.scalaquery.ql.TypeMapper._
import org.scalaquery.ql.extended.{ExtendedTable => Table}

import org.scalaquery.ql.extended.H2Driver.Implicit._

import org.scalaquery.session._

object Task extends Table[(Long, String, Date, Boolean)]("tasks") {
```

```

lazy val database = Database.forDataSource(DB.getDataSource())

def id = column[Long]("id", O PrimaryKey, O AutoInc)
def name = column[String]("name", O NotNull)
def dueDate = column[Date]("due_date")
def done = column[Boolean]("done")
def * = id ~ name ~ dueDate ~ done

def findAll = database.withSession { implicit db:Session =>
 (for(t <- this) yield t.id ~ t.name).list
}

}

```

## Exposing the datasource through JNDI

Some libraries expect to retrieve the `Datasource` reference from JNDI. You can expose any Play managed datasource via JNDI by adding this configuration

in `conf/application.conf`:

```

db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.jndiName=DefaultDS

```

**Next:** [Using the Cache](#)

## The Play cache API

Caching data is a typical optimization in modern applications, and so Play provides a global cache.

An important point about the cache is that it behaves just like a cache should: the data you just stored may just go missing.

For any data stored in the cache, a regeneration strategy needs to be put in place in case the data goes missing. This philosophy is one of the fundamentals behind Play, and is different from Java EE, where the session is expected to retain values throughout its lifetime.

The default implementation of the Cache API uses [EHCache](#).

# Importing the Cache API

Add `cache` into your dependencies list. For example, in `build.sbt`:

```
libraryDependencies ++= Seq(
 cache,
 ...
)
```

# Accessing the Cache API

The cache API is provided by the `CacheApi` object, and can be injected into your component like any other dependency. For example:

```
import play.api.cache._
import play.api.mvc._
import javax.inject.Inject

class Application @Inject()(cache: CacheApi) extends Controller {
}
```

**Note:** The API is intentionally minimal to allow several implementation to be plugged in. If you need a more specific API, use the one provided by your Cache plugin.

Using this simple API you can either store data in cache:

```
cache.set("item.key", connectedUser)
```

And then retrieve it later:

```
val maybeUser: Option[User] = cache.get[User]("item.key")
```

There is also a convenient helper to retrieve from cache or set the value in cache if it was missing:

```
val user: User = cache.getOrElse[User]("item.key") {
 User.findById(connectedUser)
}
```

You can specify an expiry duration by passing a duration, by default the duration is infinite:

```
import scala.concurrent.duration._

cache.set("item.key", connectedUser, 5.minutes)
```

To remove an item from the cache use the `remove` method:  
`cache.remove("item.key")`

# Accessing different caches

It is possible to access different caches. The default cache is called `play`, and can be configured by creating a file called `ehcache.xml`. Additional caches may be configured with different configurations, or even implementations.

If you want to access multiple different ehcache caches, then you'll need to tell Play to bind them in `application.conf`, like so:

```
play.cache.bindCaches = ["db-cache", "user-cache", "session-cache"]
```

Now to access these different caches, when you inject them, use the [NamedCache](#) qualifier on your dependency, for example:

```
import play.api.cache._
import play.api.mvc._
import javax.inject.Inject

class Application @Inject()(
 @NamedCache("session-cache") sessionCache: CacheApi
) extends Controller {

}
```

# Caching HTTP responses

You can easily create smart cached actions using standard Action composition.

**Note:** Play HTTP `Result` instances are safe to cache and reuse later.

The [Cached](#) class helps you build cached actions.

```
import play.api.cache.Cached
import javax.inject.Inject

class Application @Inject() (cached: Cached) extends Controller {

}
```

You can cache the result of an action using a fixed key like `"homePage"`.

```
def index = cached("homePage") {
 Action {
 Ok("Hello world")
 }
}
```

If results vary, you can cache each result using a different key. In this example, each user has a different cached result.

```
def userProfile = Authenticated {
 user =>
 cached(req => "profile." + user) {
 Action {
```

```

 Ok(views.html.profile(User.find(user)))
}
}
}

```

## Control caching

You can easily control what you want to cache or what you want to exclude from the cache.

You may want to only cache 200 Ok results.

```

def get(index: Int) = cached.status(_ => "/resource/" + index, 200) {
 Action {
 if (index > 0) {
 Ok(Json.obj("id" -> index))
 } else {
 NotFound
 }
 }
}

```

Or cache 404 Not Found only for a couple of minutes

```

def get(index: Int) = {
 val caching = cached
 .status(_ => "/resource/" + index, 200)
 .includeStatus(404, 600)

 caching {
 Action {
 if (index % 2 == 1) {
 Ok(Json.obj("id" -> index))
 } else {
 NotFound
 }
 }
 }
}

```

# Custom implementations

It is possible to provide a custom implementation of the [CacheApi](#) that either replaces, or sits along side the default implementation.

To replace the default implementation, you'll need to disable the default implementation by setting the following in `application.conf`:

```
play.modules.disabled += "play.api.cache.EhCacheModule"
```

Then simply implement `CacheApi` and bind it in the [DI container](#).

To provide an implementation of the cache API in addition to the default implementation, you can either create a custom qualifier, or reuse the `NamedCache` qualifier to bind the implementation.

Next: [Calling WebServices](#)

# The Play WS API

Sometimes we would like to call other HTTP services from within a Play application. Play supports this via its [WS library](#), which provides a way to make asynchronous HTTP calls.

There are two important parts to using the WS API: making a request, and processing the response. We'll discuss how to make both GET and POST HTTP requests first, and then show how to process the response from WS. Finally, we'll discuss some common use cases.

## Making a Request

To use WS, first add `ws` to your `build.sbt` file:

```
libraryDependencies += Seq(
 ws
)
```

Now any controller or component that wants to use WS will have to declare a dependency on the `WSClient`:

```
import javax.inject.Inject
import scala.concurrent.Future

import play.api.mvc._
import play.api.libs.ws._
```

```
class Application @Inject()(ws: WSClient) extends Controller {
}
```

We've called the `WSClient` instance `ws`, all the following examples will assume this name.

To build an HTTP request, you start with `ws.url()` to specify the URL.

```
val request: WSRequest = ws.url(url)
```

This returns a `WSRequest` that you can use to specify various HTTP options, such as setting headers. You can chain calls together to construct complex requests.

```
val complexRequest: WSRequest =
 request.withHeaders("Accept" -> "application/json")
 .withRequestTimeout(10000)
 .withQueryString("search" -> "play")
```

You end by calling a method corresponding to the HTTP method you want to use. This ends the chain, and uses all the options defined on the built request in the `WSRequest`.

```
val futureResponse: Future[WSResponse] = complexRequest.get()
```

This returns a `Future[WSResponse]` where the [Response](#) contains the data returned from the server.

## Request with authentication

If you need to use HTTP authentication, you can specify it in the builder, using a username, password, and an [AuthScheme](#). Valid case objects for the AuthScheme are `BASIC`, `DIGEST`, `KERBEROS`, `NONE`, `NTLM`, and `SPNEGO`.

```
ws.url(url).withAuth(user, password, WSAuthScheme.BASIC).get()
```

## Request with follow redirects

If an HTTP call results in a 302 or a 301 redirect, you can automatically follow the redirect without having to make another call.

```
ws.url(url).withFollowRedirects(true).get()
```

## Request with query parameters

Parameters can be specified as a series of key/value tuples.

```
ws.url(url).withQueryString("paramKey" -> "paramValue").get()
```

## Request with additional headers

Headers can be specified as a series of key/value tuples.

```
ws.url(url).withHeaders("headerKey" -> "headerValue").get()
```

If you are sending plain text in a particular format, you may want to define the content type explicitly.

```
ws.url(url).withHeaders("Content-Type" -> "application/xml").post(xmlString)
```

## Request with virtual host

A virtual host can be specified as a string.

```
ws.url(url).withVirtualHost("192.168.1.1").get()
```

## Request with timeout

If you wish to specify a request timeout, you can use `withRequestTimeout` to set a value in milliseconds. A value of `-1` can be used to set an infinite timeout.

```
ws.url(url).withRequestTimeout(5000).get()
```

## Submitting form data

To post url-form-encoded data a `Map[String, Seq[String]]` needs to be passed into `post`.

```
ws.url(url).post(Map("key" -> Seq("value")))
```

## Submitting JSON data

The easiest way to post JSON data is to use the [JSON](#) library.

```
import play.api.libs.json._
```

```

val data = Json.obj(
 "key1" -> "value1",
 "key2" -> "value2"
)
val futureResponse: Future[WSResponse] = ws.url(url).post(data)

```

## Submitting XML data

The easiest way to post XML data is to use XML literals. XML literals are convenient, but not very fast. For efficiency, consider using an XML view template, or a JAXB library.

```

val data = <person>
 <name>Steve</name>
 <age>23</age>
</person>
val futureResponse: Future[WSResponse] = ws.url(url).post(data)

```

# Processing the Response

Working with the [Response](#) is easily done by mapping inside the [Future](#).

The examples given below have some common dependencies that will be shown once here for brevity.

Whenever an operation is done on a `Future`, an implicit execution context must be available - this declares which thread pool the callback to the future should run in. The default Play execution context is often sufficient:

```
implicit val context = play.api.libs.concurrent.Execution.Implicits.defaultContext
```

The examples also use the following case class for serialization / deserialization:

```
case class Person(name: String, age: Int)
```

## Processing a response as JSON

You can process the response as a [JSON object](#) by calling `response.json`.

```
val futureResult: Future[String] = ws.url(url).get().map {
 response =>
 (response.json \ "person" \ "name").as[String]
}
```

The JSON library has a [useful feature](#) that will map an implicit [Reads\[T\]](#) directly to a class:

```
import play.api.libs.json._
```

```
implicit val personReads = Json.reads[Person]
```

```
val futureResult: Future[JsResult[Person]] = ws.url(url).get().map {
 response => (response.json \ "person").validate[Person]
}
```

## Processing a response as XML

You can process the response as an [XML literal](#) by calling `response.xml`.

```

val futureResult: Future[scala.xml.NodeSeq] = ws.url(url).get().map {
 response =>
 response.xml \ "message"
}

```

## Processing large responses

Calling `get()` or `post()` will cause the body of the request to be loaded into memory before the response is made available. When you are downloading with large, multi-gigabyte files, this may result in unwelcome garbage collection or even out of memory errors.

`WS` lets you use the response incrementally by using an [iteratee](#).

The `stream()` and `getStream()` methods on `WSRequest` return `Future[(WSResponseHeaders, Enumerator[Array[Byte]])]`. The enumerator contains the response body.

Here is a trivial example that uses an iteratee to count the number of bytes returned by the response:

```

import play.api.libs.iteratee._

// Make the request
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
 ws.url(url).getStream()

val bytesReturned: Future[Long] = futureResponse.flatMap {
 case (headers, body) =>
 // Count the number of bytes returned
 body |>> Iteratee.fold(0L) { (total, bytes) =>
 total + bytes.length
 }
}

```

Of course, usually you won't want to consume large bodies like this, the more common use case is to stream the body out to another location. For example, to stream the body to a file:

```

import play.api.libs.iteratee._

// Make the request
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
 ws.url(url).getStream()

val downloadedFile: Future[File] = futureResponse.flatMap {
 case (headers, body) =>
 val outputStream = new FileOutputStream(file)

 // The iteratee that writes to the output stream
 val iteratee = Iteratee.foreach[Array[Byte]] { bytes =>
 outputStream.write(bytes)
 }

```

```
// Feed the body into the iteratee
(body |>>> iteratee).andThen {
 case result =>
 // Close the output stream whether there was an error or not
 outputStream.close()
 // Get the result or rethrow the error
 result.get
 }.map(_ => file)
}
```

Another common destination for response bodies is to stream them through to a response that this server is currently serving:

```
def downloadFile = Action.async {

 // Make the request
 ws.url(url).getStream().map {
 case (response, body) =>

 // Check that the response was successful
 if (response.status == 200) {

 // Get the content type
 val contentType = response.headers.get("Content-Type").flatMap(_.headOption)
 .getOrElse("application/octet-stream")

 // If there's a content length, send that, otherwise return the body chunked
 response.headers.get("Content-Length") match {
 case Some(Seq(length)) =>
 Ok.feed(body).as(contentType).withHeaders("Content-Length" -> length)
 case _ =>
 Ok.chunked(body).as(contentType)
 }
 } else {
 BadGateway
 }
 }
}
```

`POST` and `PUT` calls require manually calling the `withMethod` method, eg:

```
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
 ws.url(url).withMethod("PUT").withBody("some body").stream()
```

# Common Patterns and Use Cases

## Chaining WS calls

Using for comprehensions is a good way to chain WS calls in a trusted environment. You should use for comprehensions together with [Future.recover](#) to handle possible failure.

```

val futureResponse: Future[WSResponse] = for {
 responseOne <- ws.url(urlOne).get()
 responseTwo <- ws.url(responseOne.body).get()
 responseThree <- ws.url(responseTwo.body).get()
} yield responseThree

futureResponse.recover {
 case e: Exception =>
 val exceptionData = Map("error" -> Seq(e.getMessage))
 ws.url(exceptionUrl).post(exceptionData)
}

```

## Using in a controller

When making a request from a controller, you can map the response to a `Future[Result]`. This can be used in combination with Play's `Action.async` action builder, as described in [Handling Asynchronous Results](#).

```

def wsAction = Action.async {
 ws.url(url).get().map { response =>
 Ok(response.body)
 }
}
status(wsAction(FakeRequest())) must_== OK

```

# Using WSClient

WSClient is a wrapper around the underlying AsyncHttpClient. It is useful for defining multiple clients with different profiles, or using a mock.

You can define a WS client directly from code without having it injected by WS, and then use it implicitly with `WS.clientUrl()`:

```

import play.api.libs.ws.ning._

implicit val sslClient = NingWSClient()
// close with sslClient.close() when finished with client
val response = WS.clientUrl(url).get()

```

NOTE: if you instantiate a NingWSClient object, it does not use the WS module lifecycle, and so will not be automatically closed in `Application.onStop`. Instead, the client must be manually shutdown using `client.close()` when processing has completed. This will release the underlying ThreadPoolExecutor used by AsyncHttpClient. Failure to close the client may result in out of memory exceptions (especially if you are reloading an application frequently in development mode).

or directly:

```
val response = sslClient.url(url).get()
```

Or use a magnet pattern to match up certain clients automatically:

```

object PairMagnet {
 implicit def fromPair(pair: (WSClient, java.net.URL)) =
 new WSRequestMagnet {
 def apply(): WSRequest = {
 val (client, netUrl) = pair
 client.url(netUrl.toString)
 }
 }
}

import scala.language.implicitConversions
import PairMagnet._

val exampleURL = new java.net.URL(url)
val response = WS.url(ws -> exampleURL).get()

By default, configuration happens in application.conf, but you can also set up the
builder directly from configuration:
import com.typesafe.config.ConfigFactory
import play.api._
import play.api.libs.ws._
import play.api.libs.ws.ning._

val configuration = Configuration.reference ++ Configuration(ConfigFactory.parseString(
 """
 |ws.followRedirects = true
 """.stripMargin))

// If running in Play, environment should be injected
val environment = Environment(new File("."), this.getClass.getClassLoader, Mode.Prod)

val parser = new WSCConfigParser(configuration, environment)
val config = new NingWSClientConfig(wsClientConfig = parser.parse())
val builder = new NingAsyncHttpClientConfigBuilder(config)

```

You can also get access to the underlying [async client](#).

```

import com.ning.http.client.AsyncHttpClient

val client: AsyncHttpClient = ws.underlying

```

This is important in a couple of cases. WS has a couple of limitations that require access to the client:

- `WS` does not support multi part form upload directly. You can use the underlying client with [RequestBuilder.addBodyPart](#).
- `WS` does not support streaming body upload. In this case, you should use the `FeedableBodyGenerator` provided by `AsyncHttpClient`.

# Configuring WS

Use the following properties in `application.conf` to configure the WS client:

- `play.ws.followRedirects`: Configures the client to follow 301 and 302 redirects (*default is true*).
- `play.ws.useProxyProperties`: To use the system http proxy settings(`http.proxyHost`, `http.proxyPort`) (*default is true*).
- `play.ws.userAgent`: To configure the User-Agent header field.
- `play.ws.compressionEnabled`: Set it to true to use gzip/deflater encoding (*default is false*).

## Configuring WS with SSL

To configure WS for use with HTTP over SSL/TLS (HTTPS), please see [Configuring WS SSL](#).

## Configuring Timeouts

There are 3 different timeouts in WS. Reaching a timeout causes the WS request to interrupt.

- `play.ws.timeout.connection`: The maximum time to wait when connecting to the remote host (*default is 120 seconds*).
- `play.ws.timeout.idle`: The maximum time the request can stay idle (connection is established but waiting for more data) (*default is 120 seconds*).
- `play.ws.timeout.request`: The total time you accept a request to take (it will be interrupted even if the remote host is still sending data) (*default is 120 seconds*).

The request timeout can be overridden for a specific connection with `withRequestTimeout()` (see “Making a Request” section).

## Configuring AsyncHttpClientConfig

The following advanced settings can be configured on the underlying `AsyncHttpClientConfig`.

Please refer to the [AsyncHttpClientConfig Documentation](#) for more information.

- `play.ws.ning.allowPoolingConnection`
- `play.ws.ning.allowSslConnectionPool`
- `play.ws.ning.ioThreadMultiplier`
- `play.ws.ning.maxConnectionsPerHost`
- `play.ws.ning.maxConnectionsTotal`
- `play.ws.ning.maxConnectionLifeTime`
- `play.ws.ning.idleConnectionInPoolTimeout`
- `play.ws.ning.webSocketIdleTimeout`
- `play.ws.ning.maxNumberOfRedirects`
- `play.ws.ning.maxRequestRetry`
- `play.ws.ning.disableUrlEncoding`

**Next:** [Connecting to OpenID services](#)

# OpenID Support in Play

OpenID is a protocol for users to access several services with a single account. As a web developer, you can use OpenID to offer users a way to log in using an account they already have, such as their [Google account](#). In the enterprise, you may be able to use OpenID to connect to a company's SSO server.

---

## The OpenID flow in a nutshell

1. The user gives you his OpenID (a URL).
2. Your server inspects the content behind the URL to produce a URL where you need to redirect the user.
3. The user confirms the authorization on his OpenID provider, and gets redirected back to your server.
4. Your server receives information from that redirect, and checks with the provider that the information is correct.

Step 1 may be omitted if all your users are using the same OpenID provider (for example if you decide to rely completely on Google accounts).

---

## Usage

To use OpenID, first add `ws` to your `build.sbt` file:

```
libraryDependencies ++= Seq(
 ws
)
```

Now any controller or component that wants to use OpenID will have to declare a dependency on the [OpenIdClient](#):

```
import javax.inject.Inject
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import play.api.libs.openid._

class Application @Inject()(openIdClient: OpenIdClient) extends Controller {
}
```

We've called the `OpenIdClient` instance `openIdClient`, all the following examples will assume this name.

---

## OpenID in Play

The OpenID API has two important functions:

- `OpenIdClient.redirectURL` calculates the URL where you should redirect the user. It involves fetching the user's OpenID page asynchronously, this is why it returns a `Future[String]`. If the OpenID is invalid, the returned `Future` will fail.
- `OpenIdClient.verifiedId` needs a `RequestHeader` and inspects it to establish the user information, including his verified OpenID. It will do a call to the OpenID server asynchronously to check the authenticity of the information, returning a future of `UserInfo`. If the information is not correct or if the server check is false (for example if the redirect URL has been forged), the returned `Future` will fail.  
If the `Future` fails, you can define a fallback, which redirects back the user to the login page or return a `BadRequest`.

Here is an example of usage (from a controller):

```
def login = Action {
 Ok(views.html.login())
}

def loginPost = Action.async { implicit request =>
 Form(single(
 "openid" -> nonEmptyText
)).bindFromRequest.fold({ error =>
 Logger.info("bad request " + error.toString)
 Future.successful(BadRequest(error.toString()))
 }, { openId =>
 openIdClient.redirectURL(openId, routes.Application.openIdCallback.absoluteURL())
 .map(url => Redirect(url))
 .recover { case t: Throwable => Redirect(routes.Application.login())}
 })
}

def openIdCallback = Action.async { implicit request =>
 openIdClient.verifiedId(request).map(info => Ok(info.id + "\n" + info.attributes))
 .recover {
 case t: Throwable =>
 // Here you should look at the error, and give feedback to the user
 Redirect(routes.Application.login())
 }
}
```

## Extended Attributes

The OpenID of a user gives you his identity. The protocol also supports getting extended attributes such as the e-mail address, the first name, or the last name.

You may request *optional* attributes and/or *required* attributes from the OpenID server. Asking for required attributes means the user cannot login to your service if he doesn't provides them.

Extended attributes are requested in the redirect URL:

```
openIdClient.redirectURL(
 openId,
 routes.Application.openIdCallback.absoluteURL(),
 Seq("email" -> "http://schema.openid.net/contact/email")
)
```

Attributes will then be available in the `UserInfo` provided by the OpenID server.

Next: [Accessing resources protected by OAuth](#)

# OAuth

**OAuth** is a simple way to publish and interact with protected data. It's also a safer and more secure way for people to give you access. For example, it can be used to access your users' data on [Twitter](#).

There are 2 very different versions of OAuth: [OAuth 1.0](#) and [OAuth 2.0](#). Version 2 is simple enough to be implemented easily without library or helpers, so Play only provides support for OAuth 1.0.

## Usage

To use OAuth, first add `ws` to your `build.sbt` file:

```
libraryDependencies += Seq(
 ws
)
```

## Required Information

OAuth requires you to register your application to the service provider. Make sure to check the callback URL that you provide, because the service provider may reject your calls if they don't match. When working locally, you can use `/etc/hosts` to fake a domain on your local machine.

The service provider will give you:

- Application ID
- Secret key
- Request Token URL
- Access Token URL
- Authorize URL

## Authentication Flow

Most of the flow will be done by the Play library.

1. Get a request token from the server (in a server-to-server call)
2. Redirect the user to the service provider, where he will grant your application rights to use his data
3. The service provider will redirect the user back, giving you a /verifier/
4. With that verifier, exchange the /request token/ for an /access token/ (server-to-server call)

Now the /access token/ can be passed to any call to access protected data.

## Example

```
object Twitter extends Controller {

 val KEY = ConsumerKey("xxxxx", "xxxxx")

 val TWITTER = OAuth[ServiceInfo(
 "https://api.twitter.com/oauth/request_token",
 "https://api.twitter.com/oauth/access_token",
 "https://api.twitter.com/oauth/authorize", KEY),
 true)

 def authenticate = Action { request =>
 request.getQueryString("oauth_verifier").map { verifier =>
 val tokenPair = sessionTokenPair(request).get
 // We got the verifier; now get the access token, store it and back to index
 TWITTER.retrieveAccessToken(tokenPair, verifier) match {
 case Right(t) => {
 // We received the authorized tokens in the OAuth object - store it before we proceed
 Redirect(routes.Application.index).withSession("token" -> t.token, "secret" -> t.secret)
 }
 case Left(e) => throw e
 }
 }.getOrElse(
 TWITTER.retrieveRequestToken("http://localhost:9000/auth") match {
 case Right(t) => {
 // We received the unauthorized tokens in the OAuth object - store it before we proceed
 Redirect(TWITTER.redirectUrl(t.token)).withSession("token" -> t.token, "secret" -> t.secret)
 }
 case Left(e) => throw e
 })
 }

 def sessionTokenPair(implicit request: RequestHeader): Option[RequestToken] = {
 for {
 token <- request.session.get("token")
 secret <- request.session.get("secret")
 } yield {
 RequestToken(token, secret)
 }
 }
}
```

```

}
object Application extends Controller {

 def timeline = Action.async { implicit request =>
 Twitter.sessionTokenPair match {
 case Some(credentials) => {
 WS.url("https://api.twitter.com/1.1/statuses/home_timeline.json")
 .sign(OAuthCalculator(Twitter.KEY, credentials))
 .get
 .map(result => Ok(result.json))
 }
 case _ => Future.successful(Redirect(routes.Twitter.authenticate))
 }
 }
}

```

**Next:** Integrating with Akka

# Integrating with Akka

Akka uses the Actor Model to raise the abstraction level and provide a better platform to build correct concurrent and scalable applications. For fault-tolerance it adopts the ‘Let it crash’ model, which has been used with great success in the telecoms industry to build applications that self-heal - systems that never stop. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

## The application actor system

Akka can work with several containers called actor systems. An actor system manages the resources it is configured to use in order to run the actors which it contains.

A Play application defines a special actor system to be used by the application. This actor system follows the application life-cycle and restarts automatically when the application restarts.

### Writing actors

To start using Akka, you need to write an actor. Below is a simple actor that simply says hello to whoever asks it to.

```

import akka.actor._

object HelloActor {
 def props = Props[HelloActor]
}

```

```

case class SayHello(name: String)
}

class HelloActor extends Actor {
 import HelloActor._

 def receive = {
 case SayHello(name: String) =>
 sender() ! "Hello, " + name
 }
}

```

This actor follows a few Akka conventions:

- The messages it sends/receives, or its *protocol*, are defined on its companion object
- It also defines a `props` method on its companion object that returns the props for creating it

## Creating and using actors

To create and/or use an actor, you need an `ActorSystem`. This can be obtained by declaring a dependency on an `ActorSystem`, like so:

```

import play.api.mvc._
import akka.actor._
import javax.inject._

```

```
import actors.HelloActor
```

```

@Singleton
class Application @Inject()(system: ActorSystem) extends Controller {

 val helloActor = system.actorOf(HelloActor.props, "hello-actor")

 ...
}

```

The `actorOf` method is used to create a new actor. Notice that we've declared this controller to be a singleton. This is necessary since we are creating the actor and storing a reference to it, if the controller was not scoped as singleton, this would mean a new actor would be created every time the controller was created, which would ultimately throw an exception because you can't have two actors in the same system with the same name.

## Asking things of actors

The most basic thing that you can do with an actor is send it a message. When you send a message to an actor, there is no response, it's fire and forget. This is also known as the *tell* pattern.

In a web application however, the *tell* pattern is often not useful, since HTTP is a protocol that has requests and responses. In this case, it is much more likely that you will want to use the *ask* pattern. The ask pattern returns a `Future`, which you can then map to your own result type.

Below is an example of using our `HelloActor` with the ask pattern:

```

import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.duration._
import akka.pattern.ask
implicit val timeout = 5.seconds

def sayHello(name: String) = Action.async {
 (helloActor ? SayHello(name)).mapTo[String].map { message =>
 Ok(message)
 }
}

```

A few things to notice:

- The ask pattern needs to be imported, and then this provides a `?>` operator on the actor.
  - The return type of the ask is a `Future[Any]`, usually the first thing you will want to do after asking actor is map that to the type you are expecting, using the `mapTo` method.
  - An implicit timeout is needed in scope - the ask pattern must have a timeout. If the actor takes longer than that to respond, the returned future will be completed with a timeout error.
- 

## Dependency injecting actors

If you prefer, you can have Guice instantiate your actors and bind actor refs to them for your controllers and components to depend on.

For example, if you wanted to have an actor that depended on the Play configuration, you might do this:

```

import akka.actor._
import javax.inject._
import play.api.Configuration

object ConfiguredActor {
 case object GetConfig
}

class ConfiguredActor @Inject()(configuration: Configuration) extends Actor {
 import ConfiguredActor._

 val config = configuration.getString("my.config").getOrElse("none")

 def receive = {
 case GetConfig =>
 sender() ! config
 }
}

```

Play provides some helpers to help providing actor bindings. These allow the actor itself to be dependency injected, and allows the actor ref for the actor to be injected into other components. To bind an actor using these helpers, create a module as described in

the dependency injection documentation, then mix in the `AkkaGuiceSupport` trait and use the `bindActor` method to bind the actor:

```
import com.google.inject.AbstractModule
import play.api.libs.concurrent.AkkaGuiceSupport

import actors.ConfiguredActor

class MyModule extends AbstractModule with AkkaGuiceSupport {
 def configure = {
 bindActor[ConfiguredActor]("configured-actor")
 }
}
```

This actor will both be named `configured-actor`, and will also be qualified with the `configured-actor` name for injection. You can now depend on the actor in your controllers and other components:

```
import play.api.mvc._
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import javax.inject._
import actors.ConfiguredActor._
import scala.concurrent.ExecutionContext
import scala.concurrent.duration._

@Singleton
class Application @Inject() (@Named("configured-actor") configuredActor: ActorRef)
 (implicit ec: ExecutionContext) extends Controller {

 implicit val timeout: Timeout = 5.seconds

 def getConfig = Action.async {
 (configuredActor ? GetConfig).mapTo[String].map { message =>
 Ok(message)
 }
 }
}
```

## Dependency injecting child actors

The above is good for injecting root actors, but many of the actors you create will be child actors that are not bound to the lifecycle of the Play app, and may have additional state passed to them.

In order to assist in dependency injecting child actors, Play utilises Guice's `AssistedInject` support.

Let's say you have the following actor, which depends configuration to be injected, plus a key:

```
import akka.actor._
```

```

import javax.inject._
import com.google.inject.persist.Assisted
import play.api.Configuration

object ConfiguredChildActor {
 case object GetConfig

 trait Factory {
 def apply(key: String): Actor
 }
}

class ConfiguredChildActor @Inject() (configuration: Configuration,
 @Assisted key: String) extends Actor {
 import ConfiguredChildActor._

 val config = configuration.getString(key).getOrElse("none")

 def receive = {
 case GetConfig =>
 sender() ! config
 }
}

```

Note that the `key` parameter is declared to be `@Assisted`, this tells that it's going to be manually provided.

We've also defined a `Factory` trait, this takes the `key`, and returns an `Actor`. We won't implement this, Guice will do that for us, providing an implementation that not only passes our `key` parameter, but also locates the `Configuration` dependency and injects that. Since the trait just returns an `Actor`, when testing this actor we can inject a factory that returns any actor, for example this allows us to inject a mocked child actor, instead of the actual one.

Now, the actor that depends on this can extend `InjectedActorSupport`, and it can depend on the factory we created:

```

import akka.actor._
import javax.inject._
import play.api.libs.concurrent.InjectedActorSupport

object ParentActor {
 case class GetChild(key: String)
}

class ParentActor @Inject() (
 childFactory: ConfiguredChildActor.Factory
) extends Actor with InjectedActorSupport {
 import ParentActor._

 def receive = {
 case GetChild(key: String) =>
 val child: ActorRef = injectedChild(childFactory(key), key)
 sender() ! child
 }
}

```

```
}
```

It uses the `injectedChild` to create and get a reference to the child actor, passing in the key.

Finally, we need to bind our actors. In our module, we use the `bindActorFactory` method

to bind the parent actor, and also bind the child factory to the child implementation:

```
import com.google.inject.AbstractModule
import play.api.libs.concurrent.AkkaGuiceSupport
```

```
import actors._
```

```
class MyModule extends AbstractModule with AkkaGuiceSupport {
 def configure = {
 bindActor[ParentActor]("parent-actor")
 bindActorFactory[ConfiguredChildActor, ConfiguredChildActor.Factory]
 }
}
```

This will get Guice to automatically bind an instance of `ConfiguredChildActor.Factory`, which will provide an instance of `Configuration` to `ConfiguredChildActor` when it's instantiated.

# Configuration

The default actor system configuration is read from the Play application configuration file. For example, to configure the default dispatcher of the application actor system, add these lines to the `conf/application.conf` file:

```
akka.actor.default-dispatcher.fork-join-executor.parallelism-max = 64
akka.actor.debug.receive = on
```

For Akka logging configuration, see [configuring logging](#).

## Changing configuration prefix

In case you want to use the `akka.*` settings for another Akka actor system, you can tell Play to load its Akka settings from another location.

```
play.akka.config = "my akka"
```

Now settings will be read from the `my-akka` prefix instead of the `akka` prefix.

```
my-akka.actor.default-dispatcher.fork-join-executor.pool-size-max = 64
my-akka.actor.debug.receive = on
```

## Built-in actor system name

By default the name of the Play actor system is `application`. You can change this via an entry in the `conf/application.conf`:

```
play.akka.actor-system = "custom-name"
```

**Note:** This feature is useful if you want to put your play application ActorSystem in an Akka cluster.

# Scheduling asynchronous tasks

You can schedule sending messages to actors and executing tasks (functions or `Runnable`). You will get a `Cancellable` back that you can call `cancel` on to cancel the execution of the scheduled operation.

For example, to send a message to the `testActor` every 300 microseconds:

```
import scala.concurrent.duration._
```

```
val cancellable = system.scheduler.schedule(
 0.microseconds, 300.microseconds, testActor, "tick")
```

**Note:** This example uses implicit conversions defined in `scala.concurrent.duration` to convert numbers to `Duration` objects with various time units.

Similarly, to run a block of code 10 milliseconds from now:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
system.scheduler.scheduleOnce(10.milliseconds) {
 file.delete()
}
```

## Using your own Actor system

While we recommend you use the built in actor system, as it sets up everything such as the correct classloader, lifecycle hooks, etc, there is nothing stopping you from using your own actor system. It is important however to ensure you do the following:

- Register a `stop hook` to shut the actor system down when Play shuts down
- Pass in the correct classloader from the Play `Environment` otherwise Akka won't be able to find your applications classes
- Ensure that either you change the location that Play reads it's akka configuration from using `play.akka.config`, or that you don't read your akka configuration from the default `akka` config, as this will cause problems such as when the systems try to bind to the same remote ports

**Next:** Internationalization

## Messages and internationalization

# Specifying languages supported by your application

A valid language code is specified by a valid ISO 639-2 language code, optionally followed by a valid ISO 3166-1 alpha-2 country code, such as `fr` or `en-US`.

To start you need to specify the languages supported by your application in the `conf/application.conf` file:

```
play.i18n.langs = ["en", "en-US", "fr"]
```

## Externalizing messages

You can externalize messages in the `conf/messages.xxx` files.

The default `conf/messages` file matches all languages. Additionally you can specify language-specific message files such as `conf/messages.fr` or `conf/messages.en-US`.

You can then retrieve messages using the `play.api.i18n.Messages` object:

```
val title = Messages("home.title")
```

All internationalization API calls take an implicit `play.api.i18n.Messages` argument retrieved from the current scope. This implicit value contains both the language to use and (essentially) the internationalized messages.

The simplest way to get such an implicit value is to use the `I18nSupport` trait. For instance you can use it as follows in your controllers:

```
import play.api.i18n.I18nSupport
class MyController(val messagesApi: MessagesApi) extends Controller with I18nSupport {
 // ...
}
```

The `I18nSupport` trait gives you an implicit `Messages` value as long as there is a `Lang` or a `RequestHeader` in the implicit scope.

**Note:** If you have a `RequestHeader` in the implicit scope, it will use the preferred language extracted from the `Accept-Language` header and matching one of the `MessagesApi` supported languages. You should add a `Messages` implicit parameter to your template like this: `@()(implicit messages: Messages)`.

**Note:** Also, Play “knows” out of the box how to inject a `MessagesApi` value (that uses the `DefaultMessagesApi` implementation), so you can just annotate your controller with the `@javax.inject.Inject` annotation and let Play automatically wire the components for you.

## Messages format

Messages are formatted using the `java.text.MessageFormat` library. For example, assuming you have message defined like:

```
files.summary = The disk {1} contains {0} file(s).
```

You can then specify parameters as:

```
Messages("files.summary", d.files.length, d.name)
```

## Notes on apostrophes

Since `Messages` uses `java.text.MessageFormat`, please be aware that single quotes are used as a meta-character for escaping parameter substitutions.

For example, if you have the following messages defined:

```
info.error=You aren't logged in!
example.formatting=When using MessageFormat, "{0}" is replaced with the first parameter.
```

you should expect the following results:

```
Messages("info.error") == "You aren't logged in!"
Messages("example.formatting") == "When using MessageFormat, '{0}' is replaced with the first parameter."
```

## Retrieving supported language from an HTTP request

You can retrieve the languages supported by a specific HTTP request:

```
def index = Action { request =>
 Ok("Languages: " + request.acceptLanguages.map(_.code).mkString(", "))
}
```

**Next:** Testing your application

## Testing your application

Writing tests for your application can be an involved process. Play offers integration with both [ScalaTest](#) and [specs2](#) and provides helpers and application stubs to make testing your application as easy as possible. For the details of using your preferred test framework with Play, see the pages on [ScalaTest](#) or [specs2](#).

- [Testing your Application with ScalaTest](#)
- [Testing your Application with specs2](#)

## Advanced testing

Play provides a number of helpers for testing specific parts of an application.

- [Testing using Guice dependency injection](#)
- [Testing with databases](#)

Next: [Testing with ScalaTest](#)

# Testing your application with ScalaTest

Writing tests for your application can be an involved process. Play provides helpers and application stubs, and ScalaTest provides an integration library, [ScalaTest + Play](#), to make testing your application as easy as possible.

## Overview

The location for tests is in the “test” folder.

You can run tests from the Play console.

- To run all tests, run `test`.
- To run only one test class, run `test-only` followed by the name of the class, i.e., `test-only my.namespace.MySpec`.
- To run only the tests that have failed, run `test-quick`.
- To run tests continually, run a command with a tilde in front, i.e. `~test-quick`.
- To access test helpers such as `FakeApplication` in console, run `test:console`.

Testing in Play is based on SBT, and a full description is available in the [testing SBT](#) chapter.

## Using ScalaTest + Play

To use *ScalaTest + Play*, you’ll need to add it to your build, by changing `build.sbt` like this:

```
libraryDependencies ++= Seq(
 "org.scalatest" %% "scalatest" % "2.2.1" % "test",
 "org.scalatestplus" %% "play" % "1.4.0-M3" % "test",
)
```

You do not need to add ScalaTest to your build explicitly. The proper version of ScalaTest will be brought in automatically as a transitive dependency of *ScalaTest + Play*. You will,

however, need to select a version of *ScalaTest + Play* that matches your Play version. You can do so by checking the [Versions, Versions, Versions](#) page for *ScalaTest + Play*.

In *ScalaTest + Play*, you define test classes by extending the `PlaySpec` trait. Here's an example:

```
import collection.mutable.Stack
import org.scalatestplus.play._

class StackSpec extends PlaySpec {

 "A Stack" must {
 "pop values in last-in-first-out order" in {
 val stack = new Stack[Int]
 stack.push(1)
 stack.push(2)
 stack.pop() mustBe 2
 stack.pop() mustBe 1
 }
 "throw NoSuchElementException if an empty stack is popped" in {
 val emptyStack = new Stack[Int]
 a [NoSuchElementException] must be thrownBy {
 emptyStack.pop()
 }
 }
 }
}
```

You can alternatively [define your own base classes](#) instead of using `PlaySpec`.

You can run your tests with Play itself, or in IntelliJ IDEA (using the [Scala plugin](#)) or in Eclipse (using the [Scala IDE](#) and the [ScalaTest Eclipse plugin](#)). Please see the [IDE page](#) for more details.

## Matchers

`PlaySpec` mixes in *ScalaTest*'s `MustMatchers`, so you can write assertions using

*ScalaTest*'s matchers DSL:

```
import play.api.test.Helpers._
```

```
"Hello world" must endWith ("world")
```

For more information, see the documentation for [MustMatchers](#).

## Mockito

You can use mocks to isolate unit tests against external dependencies. For example, if your class depends on an external `DataService` class, you can feed appropriate data to your class without instantiating a `DataService` object.

*ScalaTest* provides integration with [Mockito](#) via its `MockitoSugar` trait.

To use Mockito, mix `MockitoSugar` into your test class and then use the Mockito library to mock dependencies:

```
case class Data(retrievalDate: java.util.Date)
```

```
trait DataService {
 def findData: Data
```

```

}

import org.scalatest._
import org.scalatest.mock.MockitoSugar
import org.scalatestplus.play._

import org.mockito.Mockito._

class ExampleMockitoSpec extends PlaySpec with MockitoSugar {

 "MyService#isDailyData" should {
 "return true if the data is from today" in {
 val mockDataService = mock[DataService]
 when(mockDataService.findData) thenReturn Data(new java.util.Date())

 val myService = new MyService() {
 override def dataService = mockDataService
 }

 val actual = myService.isDailyData
 actual mustBe true
 }
 }
}

```

Mocking is especially useful for testing the public methods of classes. Mocking objects and private methods is possible, but considerably harder.

## Unit Testing Models

Play does not require models to use a particular database data access layer. However, if the application uses Anorm or Slick, then frequently the Model will have a reference to database access internally.

```

import anorm._
import anorm.SqlParser._

case class User(id: String, name: String, email: String) {
 def roles = DB.withConnection { implicit connection =>
 ...
 }
}

```

For unit testing, this approach can make mocking out the `roles` method tricky.

A common approach is to keep the models isolated from the database and as much logic as possible, and abstract database access behind a repository layer.

```
case class Role(name:String)
```

```

case class User(id: String, name: String, email:String)
trait UserRepository {
 def roles(user:User) : Set[Role]
}
class AnormUserRepository extends UserRepository {
 import anorm._
 import anorm.SqlParser._

 def roles(user:User) : Set[Role] = {
 ...
 }
}

```

and then access them through services:

```

class UserService(userRepository : UserRepository) {

 def isAdmin(user:User) : Boolean = {
 userRepository.roles(user).contains(Role("ADMIN"))
 }
}

```

In this way, the `isAdmin` method can be tested by mocking out the `UserRepository` reference and passing it into the service:

```

class UserServiceSpec extends PlaySpec with MockitoSugar {

 "UserService#isAdmin" should {
 "be true when the role is admin" in {
 val userRepository = mock[UserRepository]
 when(userRepository.roles(any[User])) thenReturn Set(Role("ADMIN"))

 val userService = new UserService(userRepository)

 val actual = userService.isAdmin(User("11", "Steve", "user@example.org"))
 actual mustBe true
 }
 }
}

```

## Unit Testing Controllers

When defining controllers as objects, they can be trickier to unit test. In Play this can be alleviated by [dependency injection](#). Another way to finesse unit testing with a controller declared as a object is to use a trait with an [explicitly typed self reference](#) to the controller:

```

trait ExampleController {
 this: Controller =>

 def index() = Action {
 Ok("ok")
 }
}

```

```
object ExampleController extends Controller with ExampleController
```

and then test the trait:

```
import scala.concurrent.Future

import org.scalatest._
import org.scalatestplus.play._

import play.api.mvc._
import play.api.test._
import play.api.test.Helpers._

class ExampleControllerSpec extends PlaySpec with Results {

 class TestController() extends Controller with ExampleController

 "Example Page#index" should {
 "should be valid" in {
 val controller = new TestController()
 val result: Future[Result] = controller.index().apply(FakeRequest())
 val bodyText: String = contentAsString(result)
 bodyText mustBe "ok"
 }
 }
}
```

When testing POST requests with, for example, JSON bodies, you won't be able to use the pattern shown above (`apply(fakeRequest)`); instead you should use

`call()` on the `testController`:

```
trait WithControllerAndRequest {
 val testController = new Controller with ApiController

 def fakeRequest(method: String = "GET", route: String = "/") = FakeRequest(method, route)
 .withHeaders(
 ("Date", "2014-10-05T22:00:00"),
 ("Authorization", "username=bob;hash=foobar==")
)
}

"REST API" should {
 "create a new user" in new WithControllerAndRequest {
 val request = fakeRequest("POST", "/user").withJsonBody(Json.parse(
 s"""{"first_name": "Alice",
 | "last_name": "Doe",
 | "credentials": {
 | "username": "alice",
 | "password": "secret"
 | }
 |}""".stripMargin))
 val apiResult = call(testController.createUser, request)
 status(apiResult) mustEqual CREATED
 val jsonResult = contentAsJson(apiResult)
```

```

 ObjectId.isValid(jsonResult \ "id").as[String]) mustBe true

 // now get the real thing from the DB and check it was created with the correct values:
 val newbie = Dao().findByUsername("alice").get
 newbie.id.toString mustEqual (jsonResult \ "id").as[String]
 newbie.firstName mustEqual "Alice"
}
}

```

# Unit Testing EssentialAction

Testing `Action` or `Filter` can require to test an `EssentialAction` ([more information about what an EssentialAction is](#))

For this, the test `Helpers.call` can be used like that:

```

class ExampleEssentialActionSpec extends PlaySpec {

 "An essential action" should {
 "can parse a JSON body" in {
 val action: EssentialAction = Action { request =>
 val value = (request.body.asJson.get \ "field").as[String]
 Ok(value)
 }

 val request = FakeRequest(POST, "/").withJsonBody(Json.parse("""{ "field": "value" }"""))

 val result = call(action, request)

 status(result) mustEqual OK
 contentAsString(result) mustEqual "value"
 }
 }
}

```

Next: [Writing functional tests with ScalaTest](#)

# Writing functional tests with ScalaTest

Play provides a number of classes and convenience methods that assist with functional testing. Most of these can be found either in the `play.api.test` package or in the `Helpers` object. The `ScalaTest + Play` integration library builds on this testing support for ScalaTest.

You can access all of Play's built-in test support and `ScalaTest + Play` with the following imports:

```
import org.scalatest._
import play.api.test._
import play.api.test.Helpers._
import org.scalatestplus.play._
```

# FakeApplication

Play frequently requires a running `Application` as context: it is usually provided from `play.api.Play.current`.

To provide an environment for tests, Play provides a `FakeApplication` class which can be configured with a different `Global` object, additional configuration, or even additional plugins.

```
val fakeApplicationWithGlobal = FakeApplication(withGlobal = Some(new GlobalSettings() {
 override def onStart(app: Application) { println("Hello world!") }
}))
```

If all or most tests in your test class need a `FakeApplication`, and they can all share the same `FakeApplication`, mix in trait `OneAppPerSuite`. You can access the `FakeApplication` from the `app` field. If you need to customize the `FakeApplication`, override `app` as shown in this example:

```
class ExampleSpec extends PlaySpec with OneAppPerSuite {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled")
)

 "The OneAppPerSuite trait" must {
 "provide a FakeApplication" in {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in {
 Play.maybeApplication mustBe Some(app)
 }
 }
}
```

If you need each test to get its own `FakeApplication`, instead of sharing the same one, use `OneAppPerTest` instead:

```
class ExampleSpec extends PlaySpec with OneAppPerTest {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override def newAppForTest(td: TestData): FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled")
)

 "The OneAppPerTest trait" must {
```

```

 "provide a new FakeApplication for each test" in {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in {
 Play.maybeApplication mustBe Some(app)
 }
 }
}

```

The reason *ScalaTest + Play* provides both `OneAppPerSuite` and `OneAppPerTest` is to allow you to select the sharing strategy that makes your tests run fastest. If you want application state maintained between successive tests, you'll need to use `OneAppPerSuite`. If each test needs a clean slate, however, you could either use `OneAppPerTest` or use `OneAppPerSuite`, but clear any state at the end of each test. Furthermore, if your test suite will run fastest if multiple test classes share the same application, you can define a master suite that mixes in `OneAppPerSuite` and nested suites that mix in `ConfiguredApp`, as shown in the example in the [documentation for ConfiguredApp](#). You can use whichever strategy makes your test suite run the fastest.

## Testing with a server

Sometimes you want to test with the real HTTP stack. If all tests in your test class can reuse the same server instance, you can mix in `OneServerPerSuite` (which will also provide a new `FakeApplication` for the suite):

```

class ExampleSpec extends PlaySpec with OneServerPerSuite {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/") => Action { Ok("ok") }
 }
)

 "test server logic" in {
 val myPublicAddress = s"localhost:$port"
 val testPaymentGatewayURL = s"http://$myPublicAddress"
 // The test payment gateway requires a callback to this server before it returns a result...
 val callbackURL = s"http://$myPublicAddress/callback"
 // await is from play.api.test.FutureAwaits
 val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
 callbackURL).get())

 response.status mustBe (OK)
 }
}

```

If all tests in your test class require separate server instance, use `OneServerPerTest` instead (which will also provide a new `FakeApplication` for the suite):

```
class ExampleSpec extends PlaySpec with OneServerPerTest {

 // Override newAppForTest if you need a FakeApplication with other than
 // default parameters.
 override def newAppForTest(testData: TestData): FakeApplication =
 new FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/") => Action { Ok("ok") }
 }
)

 "The OneServerPerTest trait" must {
 "test server logic" in {
 val myPublicAddress = s"localhost:$port"
 val testPaymentGatewayURL = s"http://$myPublicAddress"
 // The test payment gateway requires a callback to this server before it returns a result...
 val callbackURL = s"http://$myPublicAddress/callback"
 // await is from play.api.test.FutureAwaits
 val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
 callbackURL).get())

 response.status mustBe (OK)
 }
 }
}
```

The `OneServerPerSuite` and `OneServerPerTest` traits provide the port number on which the server is running as the `port` field. By default this is 19001, however you can change this either overriding `port` or by setting the system property `testserver.port`. This can be useful for integrating with continuous integration servers, so that ports can be dynamically reserved for each build.

You can also customize the `FakeApplication` by overriding `app`, as demonstrated in the previous examples.

Lastly, if allowing multiple test classes to share the same server will give you better performance than either the `OneServerPerSuite` or `OneServerPerTest` approaches, you can define a master suite that mixes in `OneServerPerSuite` and nested suites that mix in `ConfiguredServer`, as shown in the example in the [documentation for ConfiguredServer](#).

# Testing with a web browser

The `ScalaTest + Play` library builds on ScalaTest's [Selenium DSL](#) to make it easy to test your Play applications from web browsers.

To run all tests in your test class using a same browser instance, mix `OneBrowserPerSuite` into your test class. You'll also need to mix in a `BrowserFactory` trait that will provide a Selenium web driver: one of `ChromeFactory`, `FirefoxFactory`, `HtmlUnitFactory`, `InternetExplorerFactory`, `SafariFactory`.

In addition to mixing in a `BrowserFactory`, you will need to mix in a `ServerProvider` trait that provides a `TestServer`: one of `OneServerPerSuite`, `OneServerPerTest`, or `ConfiguredServer`.

For example, the following test class mixes

in `OneServerPerSuite` and `HtmlUnitFactory`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with OneBrowserPerSuite with HtmlUnitFactory {
```

```
// Override app if you need a FakeApplication with other than
// default parameters.
implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'" />" +
 "</body>" +
 "</html>"
).as("text/html")
)
 }
)
```

```
"The OneBrowserPerTest trait" must {
 "provide a web driver" in {
 go to (s"http://localhost:$port/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
}
```

If each of your tests requires a new browser instance, use `OneBrowserPerTest` instead.

As with `OneBrowserPerSuite`, you'll need to also mix in

a `ServerProvider` and `BrowserFactory`:

```
class ExampleSpec extends PlaySpec with OneServerPerTest with OneBrowserPerTest with HtmlUnitFactory {
```

```
// Override newAppForTest if you need a FakeApplication with other than
```

```

// default parameters.
override def newAppForTest(testData: TestData): FakeApplication =
 new FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'" />" +
 "</body>" +
 "</html>"
).as("text/html")
)
 }
)
}

"The OneBrowserPerTest trait" must {
 "provide a web driver" in {
 go to (s"http://localhost:$port/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
}
}
}

```

If you need multiple test classes to share the same browser instance, mix `OneBrowserPerSuite` into a master suite and `ConfiguredBrowser` into multiple nested suites. The nested suites will all share the same web browser. For an example, see the [documentation for trait ConfiguredBrowser](#).

## Running the same tests in multiple browsers

If you want to run tests in multiple web browsers, to ensure your application works correctly in all the browsers you support, you can use

traits `AllBrowsersPerSuite` or `AllBrowsersPerTest`. Both of these traits declare a `browsers` field of type `IndexedSeq[BrowserInfo]` and an abstract `sharedTests` method that takes a `BrowserInfo`. The `browsers` field indicates which browsers you want your tests to run in. The default is Chrome, Firefox, Internet Explorer, `HtmlUnit`, and Safari. You can override `browsers` if the default doesn't fit your needs. You place tests you want run in multiple browsers in the `sharedTests` method, placing the name of the browser at the end of each test name. (The browser name is

available from the `BrowserInfo` passed into `sharedTests`.) Here's an example that uses `AllBrowsersPerSuite`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerSuite {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'' />" +
 "</body>" +
 "</html>"
).as("text/html")
)
 }
)

 def sharedTests(browser: BrowserInfo) = {
 "The AllBrowsersPerSuite trait" must {
 "provide a web driver " + browser.name in {
 go to (s"http://localhost:$port/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
 }
 }
}
```

All tests declared by `sharedTests` will be run with all browsers mentioned in the `browsers` field, so long as they are available on the host system. Tests for any browser that is not available on the host system will be canceled automatically. Note that you need to append the `browser.name` manually to the test name to ensure each test in the suite has a unique name (which is required by ScalaTest). If you leave that off, you'll get a duplicate-test-name error when you run your tests.

`AllBrowsersPerSuite` will create a single instance of each type of browser and use that for all the tests declared in `sharedTests`. If you want each test to have its own, brand new browser instance, use `AllBrowsersPerTest` instead:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerTest {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
 FakeApplication(
```

```

additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'' />" +
 "</body>" +
 "</html>"
).as("text/html")
)
}
}

def sharedTests(browser: BrowserInfo) = {
 "The AllBrowsersPerTest trait" must {
 "provide a web driver" + browser.name in {
 go to (s"http://localhost:$port/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
 }
}
}
}
}

```

Although both `AllBrowsersPerSuite` and `AllBrowsersPerTest` will cancel tests for unavailable browser types, the tests will show up as canceled in the output. To can clean up the output, you can exclude web browsers that will never be available by overriding `browsers`, as shown in this example:

```

class ExampleOverrideBrowsersSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerSuite {

 override lazy val browsers =
 Vector(
 FirefoxInfo(firefoxProfile),
 ChromeInfo
)

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'' />" +
 "</body>" +
 "</html>"
).as("text/html")
)
 }
)
}

```

```

 "</body>" +
 "</html>" +
).as("text/html")
)
 }
)

```

```

def sharedTests(browser: BrowserInfo) = {
 "The AllBrowsersPerSuite trait" must {
 "provide a web driver" + browser.name in {
 go to (s"http://localhost:$port/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
 }
}

```

The previous test class will only attempt to run the shared tests with Firefox and Chrome (and cancel tests automatically if a browser is not available).

## PlaySpec

`PlaySpec` provides a convenience “super Suite” ScalaTest base class for Play tests, You get `WordSpec`, `MustMatchers`, `OptionValues`, and `WsScalaTestClient` automatically by extending `PlaySpec`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with ScalaFutures with IntegrationPatience {
```

```

// Override app if you need a FakeApplication with other than
// default parameters.
implicit override lazy val app: FakeApplication =
 FakeApplication(
 additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
 withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'" />" +
 "</body>" +
 "</html>"
).as("text/html")
)
 }
)

```

"`WsScalaTestClient`'s" must {

```

"wsUrl works correctly" in {
 val futureResult = wsUrl("/testing").get
 val body = futureResult.futureValue.body
 val expectedBody =
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'' />" +
 "</body>" +
 "</html>"
 assert(body === expectedBody)
}

"wsCall works correctly" in {
 val futureResult = wsCall(Call("get", "/testing")).get
 val body = futureResult.futureValue.body
 val expectedBody =
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title='scalatest'' />" +
 "</body>" +
 "</html>"
 assert(body === expectedBody)
}
}
}
}

```

You can mix any of the previously mentioned traits into `PlaySpec`.

## When different tests need different fixtures

In all the test classes shown in previous examples, all or most tests in the test class required the same fixtures. While this is common, it is not always the case. If different tests in the same test class need different fixtures, mix in trait `MixedFixtures`. Then give each individual test the fixture it needs using one of these no-arg functions: `AppServer`, `Chrome`, `Firefox`, `HtmlUnit`, `InternetExplorer`, or `Safari`.

You cannot mix `MixedFixtures` into `PlaySpec` because `MixedFixtures` requires a ScalaTest `fixture.Suite` and `PlaySpec` is just a regular `Suite`. If you want a convenient base class for mixed fixtures, extend `MixedPlaySpec` instead. Here's an example:

```

// MixedPlaySpec already mixes in MixedFixtures
class ExampleSpec extends MixedPlaySpec {

 // Some helper methods
 def fakeApp[A](elems: (String, String)*) = FakeApplication(additionalConfiguration = Map(elems:_*),

```

```

withRoutes = {
 case ("GET", "/testing") =>
 Action(
 Results.Ok(
 "<html>" +
 "<head><title>Test Page</title></head>" +
 "<body>" +
 "<input type='button' name='b' value='Click Me' onclick='document.title=\\\"scalatest\\\" />" +
 "</body>" +
 "</html>"
).as("text/html")
)
}
def getConfig(key: String)(implicit app: Application) = app.configuration.getString(key)

// If a test just needs a FakeApplication, use "new App":
"The App function" must {
 "provide a FakeApplication" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "make the FakeApplication available implicitly" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
 }
}

// If a test needs a FakeApplication and running TestServer, use "new Server":
"The Server function" must {
 "provide a FakeApplication" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "make the FakeApplication available implicitly" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
 }
 import Helpers._

 "send 404 on a bad request" in new Server {
 import java.net._

 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
 }
}

// If a test needs a FakeApplication, running TestServer, and Selenium
// HtmlUnit driver use "new HtmlUnit":
"The HtmlUnit function" must {
 "provide a FakeApplication" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
}

```

```

 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
"make the FakeApplication available implicitly" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled"))
{
 getConfig("ehcacheplugin") mustBe Some("disabled")
}
"start the FakeApplication" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
}
import Helpers._
"send 404 on a bad request" in new HtmlUnit {
 import java.net._
 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
}
"provide a web driver" in new HtmlUnit(fakeApp()) {
 go to ("http://localhost:" + port + "/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
}
}

// If a test needs a FakeApplication, running TestServer, and Selenium
// Firefox driver use "new Firefox":
"The Firefox function" must {
 "provide a FakeApplication" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
"make the FakeApplication available implicitly" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
}
"start the FakeApplication" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
}
import Helpers._
"send 404 on a bad request" in new Firefox {
 import java.net._
 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
}
"provide a web driver" in new Firefox(fakeApp()) {
 go to ("http://localhost:" + port + "/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
}
}

```

```

// If a test needs a FakeApplication, running TestServer, and Selenium
// Safari driver use "new Safari":
"The Safari function" must {
 "provide a FakeApplication" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "make the FakeApplication available implicitly" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
 }
 import Helpers._
 "send 404 on a bad request" in new Safari {
 import java.net._
 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
 }
 "provide a web driver" in new Safari(fakeApp()) {
 go to ("http://localhost:" + port + "/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
}

// If a test needs a FakeApplication, running TestServer, and Selenium
// Chrome driver use "new Chrome":
"The Chrome function" must {
 "provide a FakeApplication" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "make the FakeApplication available implicitly" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
 }
 import Helpers._
 "send 404 on a bad request" in new Chrome {
 import java.net._
 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
 }
 "provide a web driver" in new Chrome(fakeApp()) {
 go to ("http://localhost:" + port + "/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
}

```

```

 }

// If a test needs a FakeApplication, running TestServer, and Selenium
// InternetExplorer driver use "new InternetExplorer":
"The InternetExplorer function" must {
 "provide a FakeApplication" in new InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
 app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
 }
 "make the FakeApplication available implicitly" in new InternetExplorer(fakeApp("ehcacheplugin" ->
"disabled")) {
 getConfig("ehcacheplugin") mustBe Some("disabled")
 }
 "start the FakeApplication" in new InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
 Play.maybeApplication mustBe Some(app)
 }
 import Helpers._
 "send 404 on a bad request" in new InternetExplorer {
 import java.net._
 val url = new URL("http://localhost:" + port + "/boom")
 val con = url.openConnection().asInstanceOf[HttpURLConnection]
 try con.getResponseCode mustBe 404
 finally con.disconnect()
 }
 "provide a web driver" in new InternetExplorer(fakeApp()) {
 go to ("http://localhost:" + port + "/testing")
 pageTitle mustBe "Test Page"
 click on find(name("b")).value
 eventually { pageTitle mustBe "scalatest" }
 }
}

// If a test does not need any special fixtures, just
// write "in { () => ..."
"Any old thing" must {
 "be doable without much boilerplate" in { () =>
 1 + 1 mustEqual 2
 }
}
}

```

# Testing a template

Since a template is a standard Scala function, you can execute it from your test, and check the result:

```

"render index template" in new App {
 val html = views.html.index("Coco")

 contentAsString(html) must include ("Hello Coco")
}

```

# Testing a controller

You can call any `Action` code by providing a `FakeRequest`:

```
import scala.concurrent.Future

import org.scalatest._
import org.scalatestplus.play._

import play.api.mvc._
import play.api.test._
import play.api.test.Helpers._

class ExampleControllerSpec extends PlaySpec with Results {

 class TestController() extends Controller with ExampleController

 "Example Page#index" should {
 "should be valid" in {
 val controller = new TestController()
 val result: Future[Result] = controller.index().apply(FakeRequest())
 val bodyText: String = contentAsString(result)
 bodyText mustBe "ok"
 }
 }
}
```

Technically, you don't need `WithApplication` here, although it wouldn't hurt anything to have it.

# Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
"respond to the index Action" in new App(fakeApplication) {
 val Some(result) = route(FakeRequest(GET, "/Bob"))

 status(result) mustEqual OK
 contentType(result) mustEqual Some("text/html")
 charset(result) mustEqual Some("utf-8")
 contentAsString(result) must include ("Hello Bob")
}
```

# Testing a model

If you are using an SQL database, you can replace the database connection with an in-memory instance of an H2 database using `inMemoryDatabase`.

```
val appWithMemoryDatabase = FakeApplication(additionalConfiguration = inMemoryDatabase("test"))
"run an application" in new App(appWithMemoryDatabase) {

 val Some(macintosh) = Computer.findById(21)
```

```
macintosh.name mustEqual "Macintosh"
macintosh.introduced.value mustEqual "1984-01-24"
}
```

## Testing WS calls

If you are calling a web service, you can use `WSTestClient`. There are two calls available, `wsCall` and `wsUrl` that will take a Call or a string, respectively. Note that they expect to be called in the context of `WithApplication`.

```
wsCall(controllers.routes.Application.index()).get()
wsUrl("http://localhost:9000").get()
```

Next: [Testing with specs2](#)

# Testing your application with specs2

Writing tests for your application can be an involved process. Play provides a default test framework for you, and provides helpers and application stubs to make testing your application as easy as possible.

## Overview

The location for tests is in the “test” folder. There are two sample test files created in the test folder which can be used as templates.

You can run tests from the Play console.

- To run all tests, run `test`.
- To run only one test class, run `test-only` followed by the name of the class i.e. `test-only my.namespace.MySpec`.
- To run only the tests that have failed, run `test-quick`.
- To run tests continually, run a command with a tilde in front, i.e. `~test-quick`.
- To access test helpers such as `FakeApplication` in console, run `test:console`.

Testing in Play is based on SBT, and a full description is available in the [testing SBT chapter](#).

## Using specs2

To use Play's specs2 support, add the Play specs2 dependency to your build as a test scoped dependency:

```
libraryDependencies += specs2 % Test
```

In [specs2](#), tests are organized into specifications, which contain examples which run the system under test through various different code paths.

Specifications extend the [Specification](#) trait and are using the should/in format:  
import org.specs2.mutable.\_

```
class HelloWorldSpec extends Specification {

 "The 'Hello world' string" should {
 "contain 11 characters" in {
 "Hello world" must have size(11)
 }
 "start with 'Hello'" in {
 "Hello world" must startWith("Hello")
 }
 "end with 'world'" in {
 "Hello world" must endWith("world")
 }
 }
}
```

Specifications can be run in either IntelliJ IDEA (using the [Scala plugin](#)) or in Eclipse (using the [Scala IDE](#)). Please see the [IDE page](#) for more details.

NOTE: Due to a bug in the [presentation compiler](#), tests must be defined in a specific format to work with Eclipse:

- The package must be exactly the same as the directory path.
- The specification must be annotated with `@RunWith(classOf[JUnitRunner])`.

Here is a valid specification for Eclipse:

```
package models // this file must be in a directory called "models"

import org.specs2.mutable._
import org.specs2.runner._
import org.junit.runner._

@RunWith(classOf[JUnitRunner])
class ApplicationSpec extends Specification {
 ...
}
```

## Matchers

When you use an example, you must return an example result. Usually, you will see a statement containing a `must`:

```
"Hello world" must endWith("world")
```

The expression that follows the `must` keyword are known as `matchers`. Matchers return an example result, typically Success or Failure. The example will not compile if it does not return a result.

The most useful matchers are the `match results`. These are used to check for equality, determine the result of Option and Either, and even check if exceptions are thrown.

There are also `optional matchers` that allow for XML and JSON matching in tests.

## Mockito

Mocks are used to isolate unit tests against external dependencies. For example, if your class depends on an external `DataService` class, you can feed appropriate data to your class without instantiating a `DataService` object.

`Mockito` is integrated into specs2 as the default `mocking library`.

To use Mockito, add the following import:

```
import org.specs2.mock._
```

You can mock out references to classes like so:

```
trait DataService {
 def findData: Data
}

case class Data(retrievalDate: java.util.Date)
import org.specs2.mock._
import org.specs2.mutable._

import java.util._

class ExampleMockitoSpec extends Specification with Mockito {

 "MyService#isDailyData" should {
 "return true if the data is from today" in {
 val mockDataService = mock[DataService]
 mockDataService.findData returns Data(retrievalDate = new java.util.Date())

 val myService = new MyService() {
 override def dataService = mockDataService
 }

 val actual = myService.isDailyData
 actual must equalTo(true)
 }
 }
}
```

Mocking is especially useful for testing the public methods of classes. Mocking objects and private methods is possible, but considerably harder.

# Unit Testing Models

Play does not require models to use a particular database data access layer. However, if the application uses Anorm or Slick, then frequently the Model will have a reference to database access internally.

```
import anorm._
import anorm.SqlParser._

case class User(id: String, name: String, email: String) {
 def roles = DB.withConnection { implicit connection =>
 ...
 }
}
```

For unit testing, this approach can make mocking out the `roles` method tricky.

A common approach is to keep the models isolated from the database and as much logic as possible, and abstract database access behind a repository layer.

```
case class Role(name:String)

case class User(id: String, name: String, email:String)
trait UserRepository {
 def roles(user:User) : Set[Role]
}
class AnormUserRepository extends UserRepository {
 import anorm._
 import anorm.SqlParser._

 def roles(user:User) : Set[Role] = {
 ...
 }
}
```

and then access them through services:

```
class UserService(userRepository : UserRepository) {

 def isAdmin(user:User) : Boolean = {
 userRepository.roles(user).contains(Role("ADMIN"))
 }
}
```

In this way, the `isAdmin` method can be tested by mocking out the `UserRepository` reference and passing it into the service:

```
object UserServiceSpec extends Specification with Mockito {
```

```
 "UserService#isAdmin" should {
 "be true when the role is admin" in {
```

```

 val userRepository = mock[UserRepository]
 userRepository.roles(any[User]) returns Set(Role("ADMIN"))

 val userService = new UserService(userRepository)
 val actual = userService.isAdmin(User("11", "Steve", "user@example.org"))
 actual must beTrue
}
}
}

```

# Unit Testing Controllers

When defining controllers as objects, they can be trickier to unit test. In Play this can be alleviated by [dependency injection](#). Another way to finesse unit testing with a controller declared as a object is to use a trait with an [explicitly typed self reference](#) to the controller:

```

trait ExampleController {
 this: Controller =>

 def index() = Action {
 Ok("ok")
 }
}

object ExampleController extends Controller with ExampleController

```

and then test the trait:

```

import play.api.mvc._
import play.api.test._
import scala.concurrent.Future

object ExampleControllerSpec extends PlaySpecification with Results {

 class TestController() extends Controller with ExampleController

 "Example Page#index" should {
 "should be valid" in {
 val controller = new TestController()
 val result: Future[Result] = controller.index().apply(FakeRequest())
 val bodyText: String = contentAsString(result)
 bodyText must be equalTo "ok"
 }
 }
}

```

# Unit Testing EssentialAction

Testing `Action` or `Filter` can require to test an `EssentialAction` ([more information about what an EssentialAction is](#))

For this, the test `Helpers.call` can be used like that:

```
object ExampleEssentialActionSpec extends PlaySpecification {

 "An essential action" should {
 "can parse a JSON body" in {
 val action: EssentialAction = Action { request =>
 val value = (request.body.asJson.get \ "field").as[String]
 Ok(value)
 }

 val request = FakeRequest(POST, "/").withJsonBody(Json.parse("""{ "field": "value" }"""))

 val result = call(action, request)

 status(result) mustEqual OK
 contentAsString(result) mustEqual "value"
 }
 }
}
```

Next: [Writing functional tests with specs2](#)

# Writing functional tests with specs2

Play provides a number of classes and convenience methods that assist with functional testing. Most of these can be found either in the `play.api.test` package or in the `Helpers` object.

You can add these methods and classes by importing the following:

```
import play.api.test._
import play.api.test.Helpers._
```

## FakeApplication

Play frequently requires a running `Application` as context: it is usually provided from `play.api.Play.current`.

To provide an environment for tests, Play provides a `FakeApplication` class which can be configured with a different Global object, additional configuration, or even additional plugins.

```
val fakeApplicationWithGlobal = FakeApplication(withGlobal = Some(new GlobalSettings() {
 override def onStart(app: Application) { println("Hello world!") }
}))
```

# WithApplication

To pass in an application to an example, use `WithApplication`. An explicit `Application` can be passed in, but a default `FakeApplication` is provided for convenience.

Because `WithApplication` is a built in `Around` block, you can override it to provide your own data population:

```
abstract class WithDbData extends WithApplication {
 override def around[T: AsResult](t: => T): Result = super.around {
 setupData()
 t
 }

 def setupData() {
 // setup data
 }
}
```

"Computer model" should {

```
"be retrieved by id" in new WithDbData {
 // your test code
}
"be retrieved by email" in new WithDbData {
 // your test code
}
```

# WithServer

Sometimes you want to test the real HTTP stack from within your test, in which case you can start a test server using `WithServer`:

```
"test server logic" in new WithServer(app = fakeApplicationWithBrowser, port = testPort) {
 // The test payment gateway requires a callback to this server before it returns a result...
 val callbackURL = s"http://$myPublicAddress/callback"

 // await is from play.api.test.FutureAwaits
 val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
 callbackURL).get())

 response.status must equalTo(OK)
}
```

The `port` value contains the port number the server is running on. By default this is 19001, however you can change this either by passing the port into `WithServer`, or by setting the system property `testserver.port`. This can be useful for integrating with continuous integration servers, so that ports can be dynamically reserved for each build.

A `FakeApplication` can also be passed to the test server, which is useful for setting up custom routes and testing WS calls:

```
val appWithRoutes = FakeApplication(withRoutes = {
 case ("GET", "/") =>
 Action {
 Ok("ok")
 }
})

"test WS logic" in new WithServer(app = appWithRoutes, port = 3333) {
 await(WS.url("http://localhost:3333").get()).status must equalTo(OK)
}
```

## WithBrowser

If you want to test your application using a browser, you can use [Selenium WebDriver](#). Play will start the WebDriver for you, and wrap it in the convenient API provided by [FluentLenium](#) using `WithBrowser`. Like `WithServer`, you can change the port, `Application`, and you can also select the web browser to use:

```
val fakeApplicationWithBrowser = FakeApplication(withRoutes = {
 case ("GET", "/") =>
 Action {
 Ok(
 """
 |<html>
 |<body>
 | <div id="title">Hello Guest</div>
 | click me
 |</body>
 |</html>
 """.stripMargin) as "text/html"
 }
 case ("GET", "/login") =>
 Action {
 Ok(
 """
 |<html>
 |<body>
 | <div id="title">Hello Coco</div>
 |</body>
 |</html>
 """.stripMargin) as "text/html"
 }
})

"run in a browser" in new WithBrowser(webDriver = WebDriverFactory(HTMLUNIT), app =
fakeApplicationWithBrowser) {
 browser.goTo("/")
 // Check the page
 browser.$("#title").getTexts().get(0) must equalTo("Hello Guest")
```

```

browser.$("a").click()

browser.url must equalTo("/login")
browser.$("#title").getTexts().get(0) must equalTo("Hello Coco")
}

```

# PlaySpecification

`PlaySpecification` is an extension of `Specification` that excludes some of the mixins provided in the default specs2 specification that clash with Play helpers methods. It also mixes in the Play test helpers and types for convenience.

```

object ExamplePlaySpecificationSpec extends PlaySpecification {
 "The specification" should {

 "have access to HeaderNames" in {
 USER_AGENT must be_==(User-Agent)
 }

 "have access to Status" in {
 OK must be_==(200)
 }
 }
}

```

# Testing a view template

Since a template is a standard Scala function, you can execute it from your test, and check the result:

```

"render index template" in new WithApplication {
 val html = views.html.index("Coco")

 contentAsString(html) must contain("Hello Coco")
}

```

# Testing a controller

You can call any `Action` code by providing a `FakeRequest`:

```

"respond to the index Action" in {
 val result = controllers.Application.index()(FakeRequest())

 status(result) must equalTo(OK)
 contentType(result) must beSome("text/plain")
 contentAsString(result) must contain("Hello Bob")
}

```

Technically, you don't need `WithApplication` here, although it wouldn't hurt anything to have it.

## Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
"respond to the index Action" in new WithApplication(fakeApplication) {
 val Some(result) = route(FakeRequest(GET, "/Bob"))

 status(result) must equalTo(OK)
 contentType(result) must beSome("text/html")
 charset(result) must beSome("utf-8")
 contentAsString(result) must contain("Hello Bob")
}
```

## Testing a model

If you are using an SQL database, you can replace the database connection with an in-memory instance of an H2 database using `inMemoryDatabase`.

```
val appWithMemoryDatabase = FakeApplication(additionalConfiguration = inMemoryDatabase("test"))
"run an application" in new WithApplication(appWithMemoryDatabase) {

 val Some(macintosh) = Computer.findById(21)

 macintosh.name must equalTo("Macintosh")
 macintosh.introduced must beSome.which(_ must beEqualTo("1984-01-24"))
}
```

Next: [Testing with Guice](#)

## Testing with Guice

If you're using Guice for [dependency injection](#) then you can directly configure how components and applications are created for tests. This includes adding extra bindings or overriding existing bindings.

## GuiceApplicationBuilder

[GuiceApplicationBuilder](#) provides a builder API for configuring the dependency injection and creation of an [Application](#).

### Environment

The [Environment](#), or parts of the environment such as the root path, mode, or class loader for an application, can be specified. The configured environment will be used for loading the

application configuration, it will be used when loading modules and passed when deriving bindings from Play modules, and it will be injectable into other components.

```
import play.api.inject.guice.GuiceApplicationBuilder
val application = new GuiceApplicationBuilder()
.in(Environment(new File("path/to/app"), classLoader, Mode.Test))
.build
val application = new GuiceApplicationBuilder()
.in(new File("path/to/app"))
.in(Mode.Test)
.in(classLoader)
.build
```

## Configuration

Additional configuration can be added. This configuration will always be in addition to the configuration loaded automatically for the application. When existing keys are used the new configuration will be preferred.

```
val application = new GuiceApplicationBuilder()
.configure(Configuration("a" -> 1))
.configure(Map("b" -> 2, "c" -> "three"))
.configure("d" -> 4, "e" -> "five")
.build
```

The automatic loading of configuration from the application environment can also be overridden. This will completely replace the application configuration. For example:

```
val application = new GuiceApplicationBuilder()
.loadConfig(env => Configuration.load(env))
.build
```

## Bindings and Modules

The bindings used for dependency injection are completely configurable. The builder methods support [Play Modules and Bindings](#) and also Guice Modules.

### *Additional bindings*

Additional bindings, via Play modules, Play bindings, or Guice modules, can be added:

```
import play.api.inject.bind
val injector = new GuiceApplicationBuilder()
.bindings(new ComponentModule)
.bindings(bind[Component].to[DefaultComponent])
.injector
```

### *Override bindings*

Bindings can be overridden using Play bindings, or modules that provide bindings. For example:

```
val application = new GuiceApplicationBuilder()
.overrides(bind[Component].to[MockComponent])
```

```
.build
```

### *Disable modules*

Any loaded modules can be disabled by class name:

```
val injector = new GuiceApplicationBuilder()
.disable[ComponentModule]
.injector
```

### *Loaded modules*

Modules are automatically loaded from the classpath based on the `play.modules.enabled` configuration. This default loading of modules can be overridden. For example:

```
val injector = new GuiceApplicationBuilder()
.load(
 new play.api.inject.BuiltinModule,
 bind[Component].to[DefaultComponent]
)
.injector
```

## GuiceInjectorBuilder

[GuiceInjectorBuilder](#) provides a builder API for configuring Guice dependency injection more generally. This builder does not load configuration or modules automatically from the environment like `GuiceApplicationBuilder`, but provides a completely clean state for adding configuration and bindings. The common interface for both builders can be found in [GuiceBuilder](#). A Play [Injector](#) is created. Here's an example of instantiating a component using the injector builder:

```
import play.api.inject.guice.GuiceInjectorBuilder
import play.api.inject.bind
val injector = new GuiceInjectorBuilder()
.configure("key" -> "value")
.bindings(new ComponentModule)
.overrides(bind[Component].to[MockComponent])
.injector
```

```
val component = injector.instanceOf[Component]
```

## Overriding bindings in a functional test

Here is a full example of replacing a component with a mock component for testing. Let's start with a component, that has a default implementation and a mock implementation for testing:

```
trait Component {
```

```

def hello: String
}

class DefaultComponent extends Component {
 def hello = "default"
}

class MockComponent extends Component {
 def hello = "mock"
}

```

This component is loaded automatically using a module:

```

import play.api.{ Environment, Configuration }
import play.api.inject.Module

class ComponentModule extends Module {
 def bindings(env: Environment, conf: Configuration) = Seq(
 bind[Component].to[DefaultComponent]
)
}

```

And the component is used in a controller:

```

import play.api.mvc._
import javax.inject.Inject

class Application @Inject()(component: Component) extends Controller {
 def index() = Action {
 Ok(component.hello)
 }
}

```

To build an `Application` to use in functional tests we can simply override the binding for the component:

```

import play.api.inject.guice.GuiceApplicationBuilder
import play.api.inject.bind
val application = new GuiceApplicationBuilder()
 .overrides(bind[Component].to[MockComponent])
 .build

```

The created application can be used with the functional testing helpers for [Specs2](#) and [ScalaTest](#).

**Next:** [Testing with databases](#)

# Testing with databases

While it is possible to write functional tests using [ScalaTest](#) or [specs2](#) that test database access code by starting up a full application including the database, starting up a full

application is not often desirable, due to the complexity of having many more components started and running just to test one small part of your application.

Play provides a number of utilities for helping to test database access code that allow it to be tested with a database but in isolation from the rest of your app. These utilities can easily be used with either ScalaTest or specs2, and can make your database tests much closer to lightweight and fast running unit tests than heavy weight and slow functional tests.

## Using a database

To connect to a database, at a minimum, you just need database driver name and the url of the database, using the `Databases` companion object. For example, to connect to MySQL, you might use the following:

```
import play.api.db.Databases

val database = Databases(
 driver = "com.mysql.jdbc.Driver",
 url = "jdbc:mysql://localhost/test"
)
```

This will create a database connection pool for the MySQL `test` database running on `localhost`, with the name `default`. The name of the database is only used internally by Play, for example, by other features such as evolutions, to load resources associated with that database.

You may want to specify other configuration for the database, including a custom name, or configuration properties such as usernames, passwords and the various connection pool configuration items that Play supports, by supplying a custom name parameter and/or a custom config parameter:

```
import play.api.db.Databases

val database = Databases(
 driver = "com.mysql.jdbc.Driver",
 url = "jdbc:mysql://localhost/test",
 name = "mydatabase",
 config = Map(
 "user" -> "test",
 "password" -> "secret"
)
)
```

After using a database, since the database is typically backed by a connection pool that holds open connections and may also have running threads, you need to shut it down. This is done by calling the `shutdown` method:

```
database.shutdown()
```

Manually creating the database and shutting it down is useful if you're using a test framework that runs startup/shutdown code around each test or suite. Otherwise it's recommended that you let Play manage the connection pool for you.

## Allowing Play to manage the database for you

Play also provides a `withDatabase` helper that allows you to supply a block of code to execute with a database connection pool managed by Play. Play will ensure that it is correctly shutdown after the block of code finishes executing:

```
import play.api.db.Databases
```

```
Databases.withDatabase(
 driver = "com.mysql.jdbc.Driver",
 url = "jdbc:mysql://localhost/test"
) { database =>
 val connection = database.getConnection()
 // ...
}
```

Like the `Database.apply` factory method, `withDatabase` also allows you to pass a custom `name` and `config` map if you please.

Typically, using `withDatabase` directly from every test is an excessive amount of boilerplate code. It is recommended that you create your own helper to remove this boiler plate that your test uses. For example:

```
import play.api.db.{Database, Databases}
```

```
def withMyDatabase[T](block: Database => T) = {
 Databases.withDatabase(
 driver = "com.mysql.jdbc.Driver",
 url = "jdbc:mysql://localhost/test",
 name = "mydatabase",
 config = Map(
 "user" -> "test",
 "password" -> "secret"
)
)(block)
}
```

Then it can be easily used in each test with minimal boilerplate:

```
withMyDatabase { database =>
 val connection = database.getConnection()
 // ...
}
```

**Tip:** You can use this to externalise your test database configuration, using environment variables or system properties to configure what database to use and how to connect to it. This allows for maximum flexibility for developers to have their own environments set up the way they please, as well as for CI systems that provide particular environments that may differ to development.

## Using an in-memory database

Some people prefer not to require infrastructure such as databases to be installed in order to run tests. Play provides simple helpers to create an H2 in-memory database for these purposes:

```
import play.api.db.Databases

val database = Databases.inMemory()
```

The in-memory database can be configured, by supplying a custom name, custom URL arguments, and custom connection pool configuration. The following shows supplying the `MODE` argument to tell H2 to emulate MySQL, as well as configuring the connection pool to log all statements:

```
import play.api.db.Databases

val database = Databases.inMemory(
 name = "mydatabase",
 urlOptions = Map(
 "MODE" -> "MYSQL"
 config = Map(
 "logStatements" -> true
)
```

As with the generic database factory, ensure you always shut the in-memory database connection pool down:

```
database.shutdown()
```

If you're not using a test frameworks before/after capabilities, you may want Play to manage the in-memory database lifecycle for you, this is straightforward using `withInMemory`:

```
import play.api.db.Databases
```

```
Databases.withInMemory() { database =>
 val connection = database.getConnection()

 // ...
}
```

Like `withDatabase`, it is recommended that to reduce boilerplate code, you create your own method that wraps the `withInMemory` call:

```
import play.api.db.{Database, Databases}

def withMyDatabase[T](block: Database => T) = {
 Databases.withInMemory(
 name = "mydatabase",
 urlOptions = Map(
 "MODE" -> "MYSQL"
 config = Map(
 "logStatements" -> true
)
)
 block(database)
}
```

```
 "logStatements" -> true
)
)(block)
}
```

# Applying evolutions

When running tests, you will typically want your database schema managed for your database. If you're already using evolutions, it will often make sense to reuse the same evolutions that you use in development and production in your tests. You may also want to create custom evolutions just for testing. Play provides some convenient helpers to apply and manage evolutions without having to run a whole Play application.

To apply evolutions, you can use `applyEvolutions` from the `Evolutions` companion object:

```
import play.api.db.evolutions._
```

```
Evolutions.applyEvolutions(database)
```

This will load the evolutions from the classpath in the `evolutions/<databasename>` directory, and apply them.

After a test has run, you may want to reset the database to its original state. If you have implemented your evolutions down scripts in such a way that they will drop all the database tables, you can do this simply by calling the `cleanupEvolutions` method:

```
Evolutions.cleanupEvolutions(database)
```

## Custom evolutions

In some situations you may want to run some custom evolutions in your tests. Custom evolutions can be used by using a custom `EvolutionsReader`. The simplest of these is the `SimpleEvolutionsReader`, which is an evolutions reader that takes a preconfigured map of database names to sequences of `Evolution` scripts, and can be constructed using the convenient methods on the `SimpleEvolutionsReader` companion object. For example:

```
import play.api.db.evolutions._
```

```
Evolutions.applyEvolutions(database, SimpleEvolutionsReader.forName(
 Evolution(
 1,
 "create table test (id bigint not null, name varchar(255));",
 "drop table test;"
)
))
```

Cleaning up custom evolutions is done in the same way as cleaning up regular evolutions, using the `cleanupEvolutions` method:

```
Evolutions.cleanupEvolutions(database)
```

Note though that you don't need to pass the custom evolutions reader here, this is because the state of the evolutions is stored in the database, including the down scripts which will be used to tear down the database.

Sometimes it will be impractical to put your custom evolution scripts in code. If this is the case, you can put them in the test resources directory, under a custom path using the `ClassLoaderEvolutionsReader`. For example:

```
import play.api.db.evolutions._
```

```
Evolutions.applyEvolutions(database, ClassLoaderEvolutionsReader.forPrefix("testdatabase/"))
```

This will load evolutions, in the same structure and format as is done for development and production, from `testdatabase/evolutions/<dbname>/<n>.sql`.

## Allowing Play to manage evolutions

The `applyEvolutions` and `cleanupEvolutions` methods are useful if you're using a test framework to manage running the evolutions before and after a test. Play also provides a convenient `withEvolutions` method to manage it for you, if this lighter weight approach is desired:

```
import play.api.db.evolutions._
```

```
Evolutions.withEvolutions(database) {
 val connection = database.getConnection()
 // ...
}
```

Naturally, `withEvolutions` can be combined with `withDatabase` or `withInMemory` to reduce boilerplate code, allowing you to define a function that both instantiates the database and runs evolutions for you:

```
import play.api.db.{Database, Databases}
import play.api.db.evolutions._
```

```
def withMyDatabase[T](block: Database => T) = {
 Databases.withInMemory(
 urlOptions = Map(
 "MODE" -> "MYSQL"
),
 config = Map(
 "logStatements" -> true
)
) { database =>
 Evolutions.withEvolutions(database, SimpleEvolutionsReader.forName(
 Evolution(
 1,
 "create table test (id bigint not null, name varchar(255));",
 "drop table test;"
)
)) {
 }
}
```

```
 block(database)

 }

}
```

Having defined the custom database management method for our tests, we can now use them in a straight forward manner:

```
withMyDatabase { database =>
 val connection = database.getConnection()
 connection.prepareStatement("insert into test values (10, 'testing')").execute()

 connection.prepareStatement("select * from test where id = 10")
 .executeQuery().next() must_== true
}
```

**Next:** [Testing web service clients](#)

# Testing web service clients

A lot of code can go into writing a web service client - preparing the request, serializing and deserializing the bodies, setting the correct headers. Since a lot of this code works with strings and weakly typed maps, testing it is very important. However testing it also presents some challenges. Some common approaches include:

## Test against the actual web service

This of course gives the highest level of confidence in the client code, however it is usually not practical. If it's a third party web service, there may be rate limiting in place that prevents your tests from running (and running automated tests against a third party service is not considered being a good netizen). It may not be possible to set up or ensure the existence of the necessary data that your tests require on that service, and your tests may have undesirable side effects on the service.

## Test against a test instance of the web service

This is a little better than the previous one, however it still has a number of problems. Many third party web services don't provide test instances. It also means your tests depend on the test instance being running, meaning that test service could cause your build to fail. If the test instance is behind a firewall, it also limits where the tests can be run from.

## Mock the http client

This approach gives the least confidence in the test code - often this kind of testing amounts to testing no more than that the code does what it does, which is of no value. Tests against mock web service clients show that the code runs and does certain things, but gives no confidence as to whether anything that the code does actually correlates to valid HTTP requests being made.

## Mock the web service

This approach is a good compromise between testing against the actual web service and mocking the http client. Your tests will show that all the requests it makes are valid HTTP requests, that serialisation/deserialisation of bodies work, etc, but they will be entirely self contained, not depending on any third party services.

Play provides some helper utilities for mocking a web service in tests, making this approach to testing a very viable and attractive option.

# Testing a GitHub client

As an example, let's say you've written a GitHub client, and you want to test it. The client is very simple, it just allows you to look up the names of the public repositories:

```
import javax.inject.Inject
import play.api.libs.ws.WSClient
import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.Future

class GitHubClient(ws: WSClient, baseUrl: String) {
 @Inject def this(ws: WSClient) = this(ws, "https://api.github.com")

 def repositories(): Future[Seq[String]] = {
 ws.url(baseUrl + "/repositories").get().map { response =>
 (response.json \ "full_name").map(_.as[String])
 }
 }
}
```

Note that it takes the GitHub API base URL as a parameter - we'll override this in our tests so that we can point it to our mock server.

To test this, we want an embedded Play server that will implement this endpoint. We can do that using the `Server` `withRouter` helper in combination with the [String Interpolating Routing DSL](#):

```

import play.api.libs.json._
import play.api.mvc._
import play.api.routing.sird._
import play.core.server.Server

Server.withRouter() {
 case GET(p"/repositories") => Action {
 Results.Ok(Json.arr(Json.obj("full_name" -> "octocat/Hello-World")))
 }
} { implicit port =>

```

The `withRouter` method takes a block of code that takes as input the port number that the server starts on. By default, Play starts the server on a random free port - this means that you don't need to worry about resource contention on build servers or assigning ports to tests, but it means that your code does need to be told which port is going to be used. Now to test the GitHub client, we need a `WSClient` for it. Play provides a `WsTestClient` trait that has some factory methods for creating test clients.

The `withClient` takes an implicit port, this is handy to use in combination with the `Server.withRouter` method.

The client that the `WsTestClient.withClient` method creates here is a special client - if you give it a relative URL, then it will default the hostname to `localhost` and the port number to the port number passed in implicitly. Using this, we can simply set the base url for our GitHub client to be an empty String.

Putting it all together, we have this:

```

import play.core.server.Server
import play.api.routing.sird._
import play.api.mvc._
import play.api.libs.json._
import play.api.test._

import scala.concurrent.Await
import scala.concurrent.duration._

import org.specs2.mutable.Specification
import org.specs2.time.NoTimeConversions

object GitHubClientSpec extends Specification with NoTimeConversions {

 "GitHubClient" should {
 "get all repositories" in {
 Server.withRouter() {
 case GET(p"/repositories") => Action {
 Results.Ok(Json.arr(Json.obj("full_name" -> "octocat/Hello-World")))
 }
 } { implicit port =>
 WsTestClient.withClient { client =>
 val result = Await.result(

```

```
 new GitHubClient(client, "").repositories(), 10.seconds)
 result must_== Seq("octocat/Hello-World")
}
}
}
}
}
```

# Returning files

In the previous example, we built the json manually for the mocked service. It often will be better to capture an actual response from the service your testing, and return that. To assist with this, Play provides a `sendResource` method that allows easily creating results from files on the classpath.

So after making a request on the actual GitHub API, create a file to store it in the test resources directory. The test resources directory is either `test/resources` if you're using a Play directory layout, or `src/test/resources` if you're using a standard sbt directory layout. In this case, we'll call it `github/repositories.json`, and it will contain the following:

[

```
{
 "id": 1296269,
 "owner": {
 "login": "octocat",
 "id": 1,
 "avatar_url": "https://github.com/images/error/octocat_happy.gif",
 "gravatar_id": "",
 "url": "https://api.github.com/users/octocat",
 "html_url": "https://github.com/octocat",
 "followers_url": "https://api.github.com/users/octocat/followers",
 "following_url": "https://api.github.com/users/octocat/following{/other_user}",
 "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
 "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
 "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
 "organizations_url": "https://api.github.com/users/octocat/orgs",
 "repos_url": "https://api.github.com/users/octocat/repos",
 "events_url": "https://api.github.com/users/octocat/events{/privacy}",
 "received_events_url": "https://api.github.com/users/octocat/received_events",
 "type": "User",
 "site_admin": false
},
 "name": "Hello-World",
 "full_name": "octocat/Hello-World",
 "description": "This your first repo!",
 "private": false,
 "fork": false,

```

You may decide to modify it to suit your testing needs, for example, if your GitHub client used the URLs in the above response to make requests to other endpoints, you might remove the `https://api.github.com` prefix from them so that they too are relative, and will automatically be routed to localhost on the right port by the test client.

Now, modify the router to serve this resource:

```
import play.api.mvc._
import play.api.routing.sird._
import play.api.test._
import play.core.server.Server

Server.withRouter() {
 case GET(p"/repositories") => Action {
 Results.Ok.sendResource("github/repositories.json")
 }
} { implicit port =>
```

Note that Play will automatically set a content type of `application/json` due to the filename's extension of `.json`.

## Extracting setup code

The tests implemented so far are fine if you only have one test you want to run, but if you have many methods that you want to test, it may make more sense to extract the mock client setup code out into one helper method. For example, we could define a `withGitHubClient` method:

```
import play.api.mvc._
import play.api.routing.sird._
import play.core.server.Server
import play.api.test._

def withGitHubClient[T](block: GitHubClient => T): T = {
 Server.withRouter() {
 case GET(p"/repositories") => Action {
 Results.Ok.sendResource("github/repositories.json")
 }
 } { implicit port =>
 WsTestClient.withClient { client =>
 block(new GitHubClient(client, ""))
 }
 }
}
```

And then using it in a test looks like:

```
withGitHubClient { client =>
 val result = Await.result(client.repositories(), 10.seconds)
 result must_== Seq("octocat/Hello-World")
}
```

Next: [Logging](#)

# The Logging API

Using logging in your application can be useful for monitoring, debugging, error tracking, and business intelligence. Play provides an API for logging which is accessed through the `Logger` object and uses [Logback](#) as the logging engine.

## Logging architecture

The logging API uses a set of components that help you to implement an effective logging strategy.

### *Logger*

Your application can define `Logger` instances to send log message requests.

Each `Logger` has a name which will appear in log messages and is used for configuration.

Loggers follow a hierarchical inheritance structure based on their naming. A logger is said to be an ancestor of another logger if its name followed by a dot is the prefix of descendant logger name. For example, a logger named “com.foo” is the ancestor of a logger named “com.foo.bar.Baz.” All loggers inherit from a root logger. Logger inheritance allows you to configure a set of loggers by configuring a common ancestor.

Play applications are provided a default logger named “application” or you can create your own loggers. The Play libraries use a logger named “play”, and some third party libraries will have loggers with their own names.

### *Log levels*

Log levels are used to classify the severity of log messages. When you write a log request statement you will specify the severity and this will appear in generated log messages.

This is the set of available log levels, in decreasing order of severity.

- `OFF` - Used to turn off logging, not as a message classification.
- `ERROR` - Runtime errors, or unexpected conditions.
- `WARN` - Use of deprecated APIs, poor use of API, ‘almost’ errors, other runtime situations that are undesirable or unexpected, but not necessarily “wrong”.
- `INFO` - Interesting runtime events such as application startup and shutdown.
- `DEBUG` - Detailed information on the flow through the system.
- `TRACE` - Most detailed information.

In addition to classifying messages, log levels are used to configure severity thresholds on loggers and appenders. For example, a logger set to level `INFO` will log any request of level `INFO` or higher (`INFO`, `WARN`, `ERROR`) but will ignore requests of lower severities (`DEBUG`, `TRACE`). Using `OFF` will ignore all log requests.

## Appenders

The logging API allows logging requests to print to one or many output destinations called “appenders.” Appenders are specified in configuration and options exist for the console, files, databases, and other outputs.

Appenders combined with loggers can help you route and filter log messages. For example, you could use one appender for a logger that logs useful data for analytics and another appender for errors that is monitored by an operations team.

Note: For further information on architecture, see the [Logback documentation](#).

# Using Loggers

First import the `Logger` class and companion object:

```
import play.api.Logger
```

### *The default Logger*

The `Logger` object is your default logger and uses the name “application.” You can use it to write log request statements:

```
// Log some debug info
Logger.debug("Attempting risky calculation.")

try {
 val result = riskyCalculation

 // Log result if successful
 Logger.debug(s"Result=$result")
} catch {
 case t: Throwable =>
 // Log error with message and Throwable.
 Logger.error("Exception with riskyCalculation", t)
}
```

Using Play’s default logging configuration, these statements will produce console output similar to this:

```
[debug] application - Attempting risky calculation.
[error] application - Exception with riskyCalculation
java.lang.ArithmetricException: / by zero
 at controllers.Application$.controllers$Application$$riskyCalculation(Application.scala:32) ~[classes/:na]
 at controllers.Application$$anonfun$test$1.apply(Application.scala:18) [classes/:na]
 at controllers.Application$$anonfun$test$1.apply(Application.scala:12) [classes/:na]
```

```
at play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390) [play_2.10-2.3-M1.jar:2.3-M1]
at play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390) [play_2.10-2.3-M1.jar:2.3-M1]
```

Note that the messages have the log level, logger name, message, and stack trace if a `Throwable` was used in the log request.

### *Creating your own loggers*

Although it may be tempting to use the default logger everywhere, it's generally a bad design practice. Creating your own loggers with distinct names allows for flexible configuration, filtering of log output, and pinpointing the source of log messages.

You can create a new logger using the `Logger.apply` factory method with a name argument:

```
val accessLogger = Logger("access")
```

A common strategy for logging application events is to use a distinct logger per class using the class name. The logging API supports this with a factory method that takes a class argument:

```
val logger = Logger(this.getClass())
```

### *Logging patterns*

Effective use of loggers can help you achieve many goals with the same tool:

```
import scala.concurrent.Future
import play.api.Logger
import play.api.mvc._

trait AccessLogging {

 val accessLogger = Logger("access")

 object AccessLoggingAction extends ActionBuilder[Request] {

 def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
 accessLogger.info(s"method=${request.method} uri=${request.uri} remote-
address=${request.remoteAddress}")
 block(request)
 }
 }

 object Application extends Controller with AccessLogging {

 val logger = Logger(this.getClass())

 def index = AccessLoggingAction {
 try {
```

```

 val result = riskyCalculation
 Ok(s"Result=$result")
 } catch {
 case t: Throwable => {
 logger.error("Exception with riskyCalculation", t)
 InternalServerError("Error in calculation: " + t.getMessage())
 }
 }
}

```

This example uses [action composition](#) to define an `AccessLoggingAction` that will log request data to a logger named “access.” The `Application` controller uses this action and it also uses its own logger (named after its class) for application events. In configuration you could then route these loggers to different appenders, such as an access log and an application log.

The above design works well if you want to log request data for only specific actions. To log all requests, it’s better to use a [filter](#):

```

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import play.api.Logger
import play.api.mvc._
import play.api._

object AccessLoggingFilter extends Filter {

 val accessLogger = Logger("access")

 def apply(next: (RequestHeader) => Future[Result])(request: RequestHeader): Future[Result] = {
 val resultFuture = next(request)

 resultFuture.foreach(result => {
 val msg = s"method=${request.method} uri=${request.uri} remote-address=${request.remoteAddress}" +
 s" status=${result.header.status}";
 accessLogger.info(msg)
 })

 resultFuture
 }
}

object Global extends WithFilters(AccessLoggingFilter) {

 override def onStart(app: Application) {
 Logger.info("Application has started")
 }

 override def onStop(app: Application) {
 Logger.info("Application has stopped")
 }
}

```

In the filter version we've added the response status to the log request by logging when the `Future[Result]` completes.

# Configuration

See [configuring logging](#) for details on configuration.

Next: [Advanced topics](#)

# Handling data streams reactively

Progressive Stream Processing and manipulation is an important task in modern Web Programming, starting from chunked upload/download to Live Data Streams consumption, creation, composition and publishing through different technologies including Comet and WebSockets.

Iteratees provide a paradigm and an API allowing this manipulation, while focusing on several important aspects:

- Allowing the user to create, consume and transform streams of data.
- Treating different data sources in the same manner (Files on disk, Websockets, Chunked Http, Data Upload, ...).
- Composable: using a rich set of adapters and transformers to change the shape of the source or the consumer - construct your own or start with primitives.
- Being able to stop data being sent mid-way through, and being informed when source is done sending data.
- Non blocking, reactive and allowing control over resource consumption (Thread, Memory)

# Iteratees

An Iteratee is a consumer - it describes the way input will be consumed to produce some value. An Iteratee is a consumer that returns a value it computes after being fed enough input.

```
// an iteratee that consumes String chunks and produces an Int
Iteratee[String,Int]
```

The Iteratee interface `Iteratee[E,A]` takes two type parameters: `E`, representing the type of the Input it accepts, and `A`, the type of the calculated result.

An iteratee has one of three states: `Cont` meaning accepting more input, `Error` to indicate an error state, and `Done` which carries the calculated result. These three states are defined by the `fold` method of an `Iteratee[E, A]` interface:

```
def fold[B](folder: Step[E, A] => Future[B]): Future[B]
```

where the `Step` object has 3 states :

```
object Step {
 case class Done[+A, E](a: A, remaining: Input[E]) extends Step[E, A]
 case class Cont[E, +A](k: Input[E] => Iteratee[E, A]) extends Step[E, A]
 case class Error[E](msg: String, input: Input[E]) extends Step[E, Nothing]
}
```

The fold method defines an iteratee as one of the three mentioned states. It accepts three callback functions and will call the appropriate one depending on its state to eventually extract a required value. When calling `fold` on an iteratee you are basically saying:

- If the iteratee is in the state `Done`, then I'll take the calculated result of type `A` and what is left from the last consumed chunk of input `Input[E]` and eventually produce a `B`.
- If the iteratee is in the state `Cont`, then I'll take the provided continuation (which is accepting an input) `Input[E] => Iteratee[E, A]` and eventually produce a `B`. Note that this state provides the only way to push input into the iteratee, and get a new iteratee state, using the provided continuation function.
- If the iteratee is in the state `Error`, then I'll take the error message of type `String` and the input that caused it and eventually produce a `B`.

Depending on the state of the iteratee, `fold` will produce the appropriate `B` using the corresponding passed-in function.

To sum up, an iteratee consists of 3 states, and `fold` provides the means to do something useful with the state of the iteratee.

## Some important types in the `Iteratee` definition:

Before providing some concrete examples of iteratees, let's clarify two important types we mentioned above:

- `Input[E]` represents a chunk of input that can be either an `El[E]` containing some actual input, an `Empty` chunk or an `EOF` representing the end of the stream.  
For example, `Input[String]` can be `El("Hello!")`, `Empty`, or `EOF`
- `Future[A]` represents, as its name indicates, a future value of type `A`. This means that it is initially empty and will eventually be filled in ("redeemed") with a value of type `A`, and you can schedule a callback, among other things you can do, if you are interested in that value. A Future is a very nice primitive for synchronization and composing async calls, and is explained further at the [ScalaAsync](#) section.

## Some primitive iteratees:

By implementing the iteratee, and more specifically its `fold` method, we can now create some primitive iteratees that we can use later on.

- An iteratee in the `Done` state producing a `1 : Int` and returning `Empty` as the remaining value from the last `Input[String]`

```
val doneIteratee = new Iteratee[String,Int] {
 def fold[B](folder: Step[String,Int] => Future[B])(implicit ec: ExecutionContext) : Future[B] =
 folder(Step.Done(1, Input.Empty))
}
```

As shown above, this is easily done by calling the appropriate `apply` function, in our case that of `Done`, with the necessary information.

To use this iteratee we will make use of the `Future` that holds a promised value.

```
def folder(step: Step[String,Int]):Future[Option[Int]] = step match {
 case Step.Done(a, e) => future(Some(a))
 case Step.Cont(k) => future(None)
 case Step.Error(msg,e) => future(None)
}
```

```
val eventuallyMaybeResult: Future[Option[Int]] = doneIteratee.fold(folder)
```

```
eventuallyMaybeResult.onComplete(i => println(i))
```

of course to see what is inside the `Future` when it is redeemed we use `onComplete`

`// will eventually print 1`

```
eventuallyMaybeResult.onComplete(i => println(i))
```

There is already a built-in way allowing us to create an iteratee in the `Done` state by

providing a result and input, generalizing what is implemented above:

```
val doneIteratee = Done[String,Int](1, Input.Empty)
```

Creating a `Done` iteratee is simple, and sometimes useful, but it does not consume any input. Let's create an iteratee that consumes one chunk and eventually returns it as the computed result:

```
val consumeOneInputAndEventuallyReturnIt = new Iteratee[String,Int] {
```

```
def fold[B](folder: Step[String,Int] => Future[B])(implicit ec: ExecutionContext): Future[B] = {
 folder(Step.Cont {
 case Input.EOF => Done(0, Input.EOF) //Assuming 0 for default value
 case Input.Empty => this
 case Input.El(e) => Done(e.toInt, Input.EOF)
 })
}
```

```
def folder(step: Step[String,Int]):Future[Int] = step match {
 case Step.Done(a, _) => future(a)
 case Step.Cont(k) => k(Input.EOF).fold({
 case Step.Done(a1, _) => Future.successful(a1)
 case _ => throw new Exception("Erroneous or diverging iteratee")
 })
 case _ => throw new Exception("Erroneous iteratee")
}
```

As for `Done`, there is a built-in way to define an iteratee in the `Cont` state by providing a function that takes `Input[E]` and returns a state of `Iteratee[E, A]`:

```
val consumeOneInputAndEventuallyReturnIt = {
```

```
Cont[String,Int](in => Done(100,Input.Empty))
}
```

In the same manner there is a built-in way to create an iteratee in the `Error` state by providing an error message and an `Input[E]`

Back to the `consumeOneInputAndEventuallyReturnIt`, it is possible to create a two-step simple iteratee manually, but it becomes harder and cumbersome to create any real-world iteratee capable of consuming a lot of chunks before, possibly conditionally, it eventually returns a result. Luckily there are some built-in methods to create common iteratee shapes in the `Iteratee` object.

## Folding input:

One common task when using iteratees is maintaining some state and altering it each time input is pushed. This type of iteratee can be easily created using the `Iteratee.fold` which has the signature:

```
def fold[E, A](state: A)(f: (A, E) => A): Iteratee[E, A]
```

Reading the signature one can realize that this fold takes an initial state `A`, a function that takes the state and an input chunk `(A, E) => A` and returns an `Iteratee[E, A]` capable of consuming `E`s and eventually returning an `A`. The created iteratee will return `Done` with the computed `A` when an input `EOF` is pushed.

One example would be creating an iteratee that counts the number of bytes pushed in:

```
val inputLength: Iteratee[Array[Byte], Int] = {
 Iteratee.fold[Array[Byte], Int](0) { (length, bytes) => length + bytes.size }
}
```

Another would be consuming all input and eventually returning it:

```
val consume: Iteratee[String, String] = {
 Iteratee.fold[String, String]("") { (result, chunk) => result ++ chunk }
}
```

There is actually already a method in the `Iteratee` object that does exactly this for any `scala.TraversableLike`, called `consume`, so our example becomes:

```
val consume = Iteratee.consume[String]()
```

One common case is to create an iteratee that does some imperative operation for each chunk of input:

```
val printlnIteratee = Iteratee.foreach[String](s => println(s))
```

More interesting methods exist like `repeat`, `ignore`, and `fold1` - which is different from the preceding `fold` in that it gives one the opportunity to treat input chunks asynchronously.

Of course one should be worried now about how hard it would be to manually push input into an iteratee by folding over iteratee states over and over again. Indeed each time one has to push input into an iteratee, one has to use the `fold` function to check on its state, if

it is a `Cont` then push the input and get the new state, or otherwise return the computed result. That's when `Enumerator`s come in handy.

Next: [Enumerators](#)

# Handling data streams reactively

## Enumerators

If an iteratee represents the consumer, or sink, of input, an `Enumerator` is the source that pushes input into a given iteratee. As the name suggests, it enumerates some input into the iteratee and eventually returns the new state of that iteratee. This can be easily seen looking at the `Enumerator`'s signature:

```
trait Enumerator[E] {

 /**
 * Apply this Enumerator to an Iteratee
 */
 def apply[A](i: Iteratee[E, A]): Future[Iteratee[E, A]]

}
```

An `Enumerator[E]` takes an `Iteratee[E, A]` which is any iteratee that consumes `Input[E]` and returns a `Future[Iteratee[E, A]]` which eventually gives the new state of the iteratee.

We can go ahead and manually implement `Enumerator` instances by consequently calling the iteratee's fold method, or use one of the provided `Enumerator` creation methods. For instance we can create an `Enumerator[String]` that pushes a list of strings into an iteratee, like the following:

```
val enumerateUsers: Enumerator[String] = {
 Enumerator("Guillaume", "Sadek", "Peter", "Erwan")
}
```

Now we can apply it to the consume iteratee we created before:

```
val consume = Iteratee.consume[String]()
val newIteratee: Future[Iteratee[String, String]] = enumerateUsers(consume)
```

To terminate the iteratee and extract the computed result we pass `Input.EOF`.

An `Iteratee` carries a `run` method that does just this. It pushes an `Input.EOF` and returns a `Future[A]`, ignoring left input if any.

```
// We use flatMap since newIteratee is a promise,
```

```
// and run itself return a promise
val eventuallyResult: Future[String] = newIteratee.flatMap(i => i.run)

//Eventually print the result
eventuallyResult.onSuccess { case x => println(x) }

// Prints "GuillaumeSadekPeterErwan"
```

You might notice here that an `Iteratee` will eventually produce a result (returning a promise when calling `fold` and passing appropriate callbacks), and a `Future` eventually produces a result. Then a `Future[Iteratee[E, A]]` can be viewed as `Iteratee[E, A]`. Indeed this is what `Iteratee.flatten` does. Let's apply it to the previous example:

```
//Apply the enumerator and flatten then run the resulting iteratee
val newIteratee = Iteratee.flatten(enumerateUsers(consume))

val eventuallyResult: Future[String] = newIteratee.run
```

```
//Eventually print the result
eventuallyResult.onSuccess { case x => println(x) }
```

// Prints "GuillaumeSadekPeterErwan"

An `Enumerator` has some symbolic methods that can act as operators, which can be useful in some contexts for saving some parentheses. For example, the `|>>` method works exactly like `apply`:

```
val eventuallyResult: Future[String] = {
 Iteratee.flatten(enumerateUsers |>> consume).run
}
```

Since an `Enumerator` pushes some input into an iteratee and eventually return a new state of the iteratee, we can go on pushing more input into the returned iteratee using another `Enumerator`. This can be done either by using the `flatMap` function on `Futures` or more simply by combining `Enumerator` instances using the `andThen` method, as follows:

```
val colors = Enumerator("Red", "Blue", "Green")

val moreColors = Enumerator("Grey", "Orange", "Yellow")

val combinedEnumerator = colors.andThen(moreColors)
```

```
val eventuallyIteratee = combinedEnumerator.consume
```

As for `apply`, there is a symbolic version of the `andThen` called `>>>` that can be used to save some parentheses when appropriate:

```
val eventuallyIteratee = {
 Enumerator("Red", "Blue", "Green") >>>
 Enumerator("Grey", "Orange", "Yellow") |>>
 consume
}
```

We can also create `Enumerator`s for enumerating files contents:

```
val fileEnumerator: Enumerator[Array[Byte]] = {
 Enumerator.fromFile(new File("path/to/some/file"))
}
```

Or more generally enumerating

a `java.io.InputStream` using `Enumerator.fromStream`. It is important to note that input won't be read until the iteratee this `Enumerator` is applied on is ready to take more input.

Actually both methods are based on the more generic `Enumerator.generateM` that has the following signature:

```
def generateM[E](e: => Future[Option[E]]) = {
 ...
}
```

This method defined on the `Enumerator` object is one of the most important methods for creating `Enumerator`s from imperative logic. Looking closely at the signature, this method takes a callback function `e: => Future[Option[E]]` that will be called each time the iteratee this `Enumerator` is applied to is ready to take some input.

It can be easily used to create an `Enumerator` that represents a stream of time values every 100 millisecond using the opportunity that we can return a promise, like the following:

```
Enumerator.generateM {
 Promise.timeout(Some(new Date), 100 milliseconds)
}
```

In the same manner we can construct an `Enumerator` that would fetch a url every some time using the `WS` api which returns, not surprisingly a `Future`

Combining this, callback Enumerator, with an imperative `Iteratee.foreach` we can `println` a stream of time values periodically:

```
val timeStream = Enumerator.generateM {
 Promise.timeout(Some(new Date), 100 milliseconds)
}
```

```
val printlnSink = Iteratee.foreach[Date](date => println(date))
```

```
timeStream |> printlnSink
```

Another, more imperative, way of creating an `Enumerator` is by using `Concurrent.unicast` which once it is ready will give a `Channel` interface on which defined methods `push` and `end`:

```
val enumerator = Concurrent.unicast[String](onStart = channel => {
 channel.push("Hello")
 channel.push("World")
})
```

```
enumerator |> Iteratee.foreach(println)
```

The `onStart` function will be called each time the `Enumerator` is applied to an `Iteratee`. In some applications, a chatroom for instance, it makes sense to assign the `enumerator` to a synchronized global value (using STMs for example) that will contain a list of listeners. `Concurrent.unicast` accepts two other functions, `onComplete` and `onError`.

One more interesting method is the `interleave` or `>-` method which as the name says, interleaves two `Enumerators`. For reactive `Enumerator`s `Input` will be passed as it happens from any of the interleaved `Enumerator`s

## Enumerators à la carte

Now that we have several interesting ways of creating `Enumerator`s, we can use these together with composition methods `andThen` / `>>` and `interleave` / `>-` to compose `Enumerator`s on demand.

Indeed one interesting way of organizing a streamful application is by creating primitive `Enumerator`s and then composing a collection of them. Let's imagine doing an application for monitoring systems:

```
object AvailableStreams {

 val cpu: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

 val memory: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

 val threads: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

 val heap: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

}

val physicalMachine = AvailableStreams.cpu >- AvailableStreams.memory
val jvm = AvailableStreams.threads >- AvailableStreams.heap
```

```
def usersWidgetsComposition(prefs: Preferences) = {
 // do the composition dynamically
}
```

Now, it is time to adapt and transform `Enumerator`s and `Iteratee`s using ... `Enumeratee`s!

Next: [Enumeratees](#)

## Handling data streams reactively

## The realm of `Enumeratee`s

'Enumeratee' is a very important component in the iteratees API. It provides a way to adapt and transform streams of data. An `Enumeratee` that might sound familiar is the `Enumeratee.map`.

Starting with a simple problem, consider the following `Iteratee`:

```
val sum: Iteratee[Int,Int] = Iteratee.fold[Int,Int](0){ (s,e) => s + e }
```

This `Iteratee` takes `Int` objects as input and computes their sum. Now if we have an `Enumerator` like the following:

```
val strings: Enumerator[String] = Enumerator("1","2","3","4")
```

Then obviously we can not apply the `strings: Enumerator[String]` to an `Iteratee[Int, Int]`. What we need is transform each `String` to the corresponding `Int` so that the source and the consumer can be fit together. This means we either have to adapt the `Iteratee[Int, Int]` to be `Iteratee[String, Int]`, or adapt the `Enumerator[String]` to be rather an `Enumerator[Int]`.

An `Enumeratee` is the right tool for doing that. We can create

an `Enumeratee[String, Int]` and adapt our `Iteratee[Int, Int]` using it:

```
//create an Enumeratee using the map method on Enumerator
```

```
valToInt: Enumerator[String,Int] = Enumerator.map[String]{ s => s.toInt }
```

```
val adaptedIteratee: Iteratee[String,Int] = toInt.transform(sum)
```

//this works!

```
strings |>> adaptedIteratee
```

There is a symbolic alternative to the `transform` method, `&>>` which we can use in our previous example:

```
strings |>> toInt &>> sum
```

The `map` method will create an 'Enumeratee' that uses a provided `From => To` function to map the input from the `From` type to the `To` type. We can also adapt the `Enumerator`:

```
val adaptedEnumerator: Enumerator[Int] = strings.through(toInt)
```

//this works!

```
adaptedEnumerator |>> sum
```

Here too, we can use a symbolic version of the `through` method:

```
strings &>> toInt |>> sum
```

Let's have a look at the `transform` signature defined in the `Enumeratee` trait:

```
trait Enumeratee[From, To] {
 def transform[A](inner: Iteratee[To, A]): Iteratee[From, A] = ...
}
```

This is a fairly simple signature, and is the same for `through` defined on an `Enumerator`:

```
trait Enumerator[E] {
 def through[To](enumeratee: Enumeratee[E, To]): Enumerator[To]
}
```

The `transform` and `through` methods on an `Enumeratee` and `Enumerator`, respectively, both use the `apply` method on `Enumeratee`, which has a slightly more sophisticated signature:

```
trait Enumeratee[From, To] {
 def apply[A](inner: Iteratee[To, A]): Iteratee[From, Iteratee[To, A]] = ...
}
```

```
}
```

Indeed, an `Enumeratee` is more powerful than just transforming an `Iteratee` type. It really acts like an adapter in that you can get back your original `Iteratee` after pushing some different input through an `Enumeratee`. So in the previous example, we can get back the original `Iteratee[Int, Int]` to continue pushing some `Int` objects in:

```
val sum:Iteratee[Int,Int] = Iteratee.fold[Int,Int](0){ (s,e) => s + e }
```

```
//create an Enumeratee using the map method on Enumeratee
```

```
val toInt: Enumeratee[String,Int] = Enumeratee.map[String]{ s => s.toInt }
```

```
val adaptedIteratee: Iteratee[String,Iteratee[Int,Int]] = toInt(sum)
```

```
// pushing some strings
```

```
val afterPushingStrings: Future[Iteratee[String,Iteratee[Int,Int]]] = {
```

```
 Enumerator("1","2","3","4") |> adaptedIteratee
```

```
}
```

```
val flattenAndRun:Future[Iteratee[Int,Int]] = Iteratee.flatten(afterPushingStrings).run
```

```
val originalIteratee = Iteratee.flatten(flattenAndRun)
```

```
val moreInts: Future[Iteratee[Int,Int]] = Enumerator(5,6,7) |> originalIteratee
```

```
val sumFuture:Future[Int] = Iteratee.flatten(moreInts).run
```

```
sumFuture onSuccess {
```

```
 case s => println(s)// eventually prints 28
```

```
}
```

That's why we call the adapted (original) `Iteratee` 'inner' and the resulting `Iteratee` 'outer'.

Now that the `Enumeratee` picture is clear, it is important to know that `transform` drops the left input of the inner `Iteratee` when it is `Done`. This means that if we use `Enumeratee.map` to transform input, if the inner `Iteratee` is `Done` with some left transformed input, the `transform` method will just ignore it.

That might have seemed like a bit too much detail, but it is useful for grasping the model.

Back to our example on `Enumeratee.map`, there is a more general

method `Enumeratee.mapInput` which, for example, gives the opportunity to return an `EOF` on some signal:

```
val toIntOrEnd: Enumeratee[String,Int] = Enumeratee.mapInput[String] {
 case Input.EI("end") => Input.EOF
 case other => other.map(e => e.toInt)
}
```

`Enumeratee.map` and `Enumeratee.mapInput` are pretty straightforward, they operate on a per chunk basis and they convert them. Another useful `Enumeratee` is the `Enumeratee.filter`:

```
def filter[E](predicate: E => Boolean): Enumeratee[E, E]
```

The signature is pretty obvious, `Enumeratee.filter` creates an `Enumeratee[E, E]` and it will test each chunk of input using the provided `predicate: E => Boolean` and it passes it along to the inner (adapted) iteratee if it satisfies the predicate:

```
val numbers = Enumerator(1,2,3,4,5,6,7,8,9,10)
```

```
val onlyOdds = Enumeratee.filter[Int](i => i % 2 != 0)
```

```
numbers.through(onlyOdds) |>> sum
```

There are methods, such

`as` `Enumeratee.collect`, `Enumeratee.drop`, `Enumeratee.dropWhile`, `Enumeratee.take`, `Enumeratee.takeWhile`, which work on the same principle.

Let try to use the `Enumeratee.take` on an Input of chunks of bytes:

```
// computes the size in bytes
val fillInMemory: Iteratee[Array[Byte], Array[Byte]] = {
 Iteratee.consume[Array[Byte]]()
}
```

```
val limitTo100: Enumeratee[Array[Byte], Array[Byte]] = {
 Enumeratee.take[Array[Byte]](100)
}
```

```
val limitedFillInMemory: Iteratee[Array[Byte], Array[Byte]] = {
 limitTo100 &>> fillInMemory
}
```

It looks good, but how many bytes are we taking? What would ideally limit the size, in bytes, of loaded input. What we do above is to limit the number of chunks instead, whatever the size of each chunk is. It seems that the `Enumeratee.take` is not enough here since it has no information about the type of input (in our case an `Array[Byte]`) and this is why it can't count what's inside.

Luckily there is a `Traversable` object that offers a set of methods for creating `Enumeratee` instances for Input types that are `TraversableLike`.

An `Array[Byte]` is `TraversableLike` and so we can use `Traversable.take`:

```
val fillInMemory: Iteratee[Array[Byte], Array[Byte]] = {
 Iteratee.consume[Array[Byte]]()
}
```

```
val limitTo100: Enumeratee[Array[Byte], Array[Byte]] = {
 Traversable.take[Array[Byte]](100)
}
```

```
// We are sure not to get more than 100 bytes loaded into memory
val limitedFillInMemory: Iteratee[Array[Byte], Array[Byte]] = {
 limitTo100 &>> fillInMemory
}
```

Other `Traversable` methods exist

including `Traversable.takeUpTo`, `Traversable.drop`.

Finally, you can compose different `Enumeratee` instances using the `compose` method, which has the symbolic equivalent `><`. Note that any left input on the `Done` of the

composed `Enumeratee` instances will be dropped. However, if you use `composeConcat` aliased `>+>`, any left input will be concatenated.

Next: [HTTP architecture](#)

# Introduction to Play HTTP API

## What is EssentialAction?

The `EssentialAction` is the new simpler type replacing the old `Action[A]`. To understand `EssentialAction` we need to understand the Play architecture.

The core of Play2 is really small, surrounded by a fair amount of useful APIs, services and structure to make Web Programming tasks easier.

Basically, Play2 is an API that abstractly have the following type:

`RequestHeader -> Array[Byte] -> Result`

The above `computation` takes the request header `RequestHeader`, then takes the request body as `Array[Byte]` and produces a `Result`.

Now this type presumes putting request body entirely into memory (or disk), even if you only want to compute a value out of it, or better forward it to a storage service like Amazon S3.

We rather want to receive request body chunks as a stream and be able to process them progressively if necessary.

What we need to change is the second arrow to make it receive its input in chunks and eventually produce a result. There is a type that does exactly this, it is called `Iteratee` and takes two type parameters.

`Iteratee[E, R]` is a type of `arrow` that will take its input in chunks of type `E` and eventually return `R`. For our API we need an `Iteratee` that takes chunks of `Array[Byte]` and eventually return a `Result`. So we slightly modify the type to be:

`RequestHeader -> Iteratee[Array[Byte], Result]`

For the first arrow, we are simply using the `Function[From, To]` which could be type aliased with `=>`:

`RequestHeader => Iteratee[Array[Byte], Result]`

Now if I define an infix type alias for `Iteratee[E, R]`:

`type ==> [E, R] = Iteratee[E, R]` then I can write the type in a funnier way:

`RequestHeader => Array[Byte] ==> Result`

And this should read as: Take the request headers, take chunks of `Array[Byte]` which represent the request body and eventually return a `Result`. This exactly how the `EssentialAction` type is defined:

`trait EssentialAction extends (RequestHeader => Iteratee[Array[Byte], Result])`

The `Result` type, on the other hand, can be abstractly thought of as the response headers and the body of the response:

`case class Result(headers: ResponseHeader, body: Array[Byte])`

But, what if we want to send the response body progressively to the client without filling it entirely into memory. We need to improve our type. We need to replace the body type from an `Array[Byte]` to something that produces chunks of `Array[Byte]`.

We already have a type for this and is called `Enumerator[E]` which means that it is capable of producing chunks of `E`, in our case `Enumerator[Array[Byte]]`:

`case class Result(headers: ResponseHeaders, body: Enumerator[Array[Byte]])`

If we don't have to send the response progressively we still can send the entire body as a single chunk.

We can stream and write any type of data to socket as long as it is convertible to `Array[Byte]`, that is what `Writable[E]` insures for a given type 'E':

`case class Result[E](headers: ResponseHeaders, body: Enumerator[E])(implicit writable: Writable[E])`

## Bottom Line

The essential Play2 HTTP API is quite simple:

`RequestHeader -> Iteratee[Array[Byte], Result]`

or the funnier

`RequestHeader => Array[Byte] ==> Result`

Which reads as the following: Take the `RequestHeader` then take chunks of `Array[Byte]` and return a response. A response consists of `ResponseHeaders` and a body which is chunks of values convertible to `Array[Byte]` to be written to the socket represented in the `Enumerator[E]` type.

**Next: HTTP filters**

## Filters

Play provides a simple filter API for applying global filters to each request.

# Filters vs action composition

The filter API is intended for cross cutting concerns that are applied indiscriminately to all routes. For example, here are some common use cases for filters:

- Logging/metrics collection
- [GZIP encoding](#)
- [Security headers](#)

In contrast, [action composition](#) is intended for route specific concerns, such as authentication and authorisation, caching and so on. If your filter is not one that you want applied to every route, consider using action composition instead, it is far more powerful. And don't forget that you can create your own action builders that compose your own custom defined sets of actions to each route, to minimise boilerplate.

## A simple logging filter

The following is a simple filter that times and logs how long a request takes to execute in Play framework:

```
import play.api.Logger
import play.api.mvc._
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits.defaultContext

class LoggingFilter extends Filter {

 def apply(nextFilter: RequestHeader => Future[Result])
 (requestHeader: RequestHeader): Future[Result] = {

 val startTime = System.currentTimeMillis

 nextFilter(requestHeader).map { result =>

 val endTime = System.currentTimeMillis
 val requestTime = endTime - startTime

 Logger.info(s"${requestHeader.method} ${requestHeader.uri} " +
 s"took ${requestTime}ms and returned ${result.header.status}")

 result.withHeaders("Request-Time" -> requestTime.toString)
 }
 }
}
```

Let's understand what's happening here. The first thing to notice is the signature of the `apply` method. It's a curried function, with the first parameter, `nextFilter`, being a function that takes a request header and produces a result, and the second parameter, `requestHeader`, being the actual request header of the incoming request. The `nextFilter` parameter represents the next action in the filter chain. Invoking it will cause the action to be invoked. In most cases you will probably want to invoke this at some point in your future. You may decide to not invoke it if for some reason you want to block the request.

We save a timestamp before invoking the next filter in the chain. Invoking the next filter returns a `Future[Result]` that will be redeemed eventually. Take a look at the [Handling asynchronous results](#) chapter for more details on asynchronous results. We then manipulate the `Result` in the `Future` by calling the `map` method with a closure that takes a `Result`. We calculate the time it took for the request, log it and send it back to the client in the response headers by calling `result.withHeaders("Request-Time" -> requestTime.toString)`.

## Using filters

The simplest way to use a filter is to provide an implementation of the `HttpFilters` trait in the root package:

```
import javax.inject.Inject
import play.api.http.HttpFilters
import play.filters.gzip.GzipFilter
```

```
class Filters @Inject() (
 gzip: GzipFilter,
 log: LoggingFilter
) extends HttpFilters {

 val filters = Seq(gzip, log)
}
```

If you want to have different filters in different environments, or would prefer not putting this class in the root package, you can configure where Play should find the class by setting `play.http.filters` in `application.conf` to the fully qualified class name of the class. For example:

```
play.http.filters=com.example.MyFilters
```

## Where do filters fit in?

Filters wrap the action after the action has been looked up by the router. This means you cannot use a filter to transform a path, method or query parameter to impact the router. However you can direct the request to a different action by invoking that action directly from the filter, though be aware that this will bypass the rest of the filter chain. If you do need to

modify the request before the router is invoked, a better way to do this would be to place your logic in `Global.onRouteRequest` instead.

Since filters are applied after routing is done, it is possible to access routing information from the request, via the `tags` map on the `RequestHeader`. For example, you might want to log the time against the action method. In that case, you might update the `logTime` method to look like this:

```
import play.api.mvc.{Result, RequestHeader, Filter}
import play.api.{Logger, Routes}
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits.defaultContext

object LoggingFilter extends Filter {
 def apply(nextFilter: RequestHeader => Future[Result])
 (requestHeader: RequestHeader): Future[Result] = {

 val startTime = System.currentTimeMillis

 nextFilter(requestHeader).map { result =>

 val action = requestHeader.tags(Routes.ROUTE_CONTROLLER) +
 "." + requestHeader.tags(Routes.ROUTE_ACTION_METHOD)
 val endTime = System.currentTimeMillis
 val requestTime = endTime - startTime

 Logger.info(s"${action} took ${requestTime}ms" +
 s" and returned ${result.header.status}")

 result.withHeaders("Request-Time" -> requestTime.toString)
 }
 }
}
```

Routing tags are a feature of the Play router. If you use a custom router, or return a custom action in `Global.onRouteRequest`, these parameters may not be available.

## More powerful filters

Play provides a lower level filter API called `EssentialFilter` which gives you full access to the body of the request. This API allows you to wrap `EssentialAction` with another action. Here is the above filter example rewritten as an `EssentialFilter`:

```
import play.api.Logger
import play.api.mvc._
import play.api.libs.concurrent.Execution.Implicits.defaultContext

class LoggingFilter extends EssentialFilter {
 def apply(nextFilter: EssentialAction) = new EssentialAction {
 def apply(requestHeader: RequestHeader) = {

 val startTime = System.currentTimeMillis
```

```

nextFilter(requestHeader).map { result =>
 val endTime = System.currentTimeMillis
 val requestTime = endTime - startTime

 Logger.info(s"${requestHeader.method} ${requestHeader.uri}" +
 s" took ${requestTime}ms and returned ${result.header.status}")
 result.withHeaders("Request-Time" -> requestTime.toString)
}

}
}
}
}

```

The key difference here, apart from creating a new `EssentialAction` to wrap the passed in `next` action, is when we invoke `next`, we get back an `Iteratee`. You could wrap this in an `Enumeratee` to do some transformations if you wished. We then `map` the result of the iteratee and thus handle it.

Although it may seem that there are two different filter APIs, there is only one, `EssentialFilter`. The simpler `Filter` API in the earlier examples extends `EssentialFilter`, and implements it by creating a new `EssentialAction`. The passed in callback makes it appear to skip the body parsing by creating a promise for the `Result`, while the body parsing and the rest of the action are executed asynchronously.

Next: [HTTP request handlers](#)

# HTTP Request Handlers

Play provides a range of abstractions for routing requests to actions, providing routers and filters to allow most common needs. Sometimes however an application will have more advanced needs that aren't met by Play's abstractions. When this is the case, applications can provide custom implementations of Play's lowest level HTTP pipeline API, the `HttpRequestHandler`.

Providing a custom `HttpRequestHandler` should be a last course of action. Most custom needs can be met through implementing a custom router or a [filter](#).

## Implementing a custom request handler

The `HttpRequestHandler` trait has one method to be implemented, `handlerForRequest`. This takes the request to get a handler for, and returns a tuple of a `RequestHeader` and a `Handler`.

The reason why a request header is returned is so that information can be added to the request, for example, routing information. In this way, the router is able to tag requests with routing information, such as which route matched the request, which can be useful for monitoring or even for injecting cross cutting functionality.

A very simple request handler that simply delegates to a router might look like this:

```
import javax.inject.Inject
import play.api.http._
import play.api.mvc._
import play.api.routing.Router

class SimpleHttpRequestHandler @Inject()(router: Router) extends HttpRequestHandler {
 def handlerForRequest(request: RequestHeader) = {
 router.routes.lift(request) match {
 case Some(handler) => (request, handler)
 case None => (request, Action(Results.NotFound))
 }
 }
}
```

## Extending the default request handler

In most cases you probably won't want to create a new request handler from scratch, you'll want to build on the default one. This can be done by extending [DefaultHttpRequestHandler](#). The default request handler provides a number of methods that can be overridden, this allows you to implement your custom functionality without reimplementing the code to tag requests, handle errors, etc.

One use case for a custom request handler may be that you want to delegate to a different router, depending on what host the request is for. Here is an example of how this might be done:

```
import javax.inject.Inject
import play.api.http._
import play.api.mvc.RequestHeader

class VirtualHostRequestHandler @Inject()(errorHandler: HttpErrorHandler,
 configuration: HttpConfiguration, filters: HttpFilters,
 fooRouter: foo.Routes, barRouter: bar.Routes
) extends DefaultHttpRequestHandler(
 fooRouter, errorHandler, configuration, filters
) {

 override def routeRequest(request: RequestHeader) = {
 request.host match {
```

```
 case "foo.example.com" => fooRouter.routes.lift(request)
 case "bar.example.com" => barRouter.routes.lift(request)
 case _ => super.routeRequest(request)
 }
}
```

# Configuring the http request handler

A custom http handler can be supplied by creating a class in the root package called `RequestHandler` that implements `HttpRequestHandler`.

If you don't want to place your request handler in the root package, or if you want to be able to configure different request handlers for different environments, you can do this by configuring the `play.http.requestHandler` configuration property in `application.conf`:

```
play.http.requestHandler = "com.example.RequestHandler"
```

## Performance notes

The http request handler that Play uses if none is configured is one that delegates to the legacy `GlobalSettings` methods. This may have a performance impact as it will mean your application has to do many lookups out of Guice to handle a single request. If you are not using a `Global` object, then you don't need this, instead you can configure Play to use the default http request handler:

```
play.http.requestHandler = "play.api.http.DefaultHttpRequestHandler"
```

Next: [Dependency injection](#)

# Runtime Dependency Injection

Dependency injection is a way that you can separate your components so that they are not directly dependent on each other, rather, they get injected into each other.

Out of the box, Play provides runtime dependency injection based on [JSR 330](#). Runtime dependency injection is so called because the dependency graph is created, wired and validated at runtime. If a dependency cannot be found for a particular component, you won't get an error until you run your application. In contrast, Play also supports [compile time dependency injection](#), where errors in the dependency graph are detected and thrown at compile time.

The default JSR 330 implementation that comes with Play is [Guice](#), but other JSR 330 implementations can be plugged in.

# Declaring dependencies

If you have a component, such as a controller, and it requires some other components as dependencies, then this can be declared using the `@Inject` annotation.

The `@Inject` annotation can be used on fields or on constructors, we recommend that you use it on constructors, for example:

```
import javax.inject._
import play.api.libs.ws._
```

```
class MyComponent @Inject() (ws: WSClient) {
 // ...
}
```

Note that the `@Inject` annotation must come after the class name but before the constructor parameters, and must have parenthesis.

# Dependency injecting controllers

There are two ways to make Play use dependency injected controllers.

## Injected routes generator

By default, Play will generate a static router, that assumes that all actions are static methods. By configuring Play to use the injected routes generator, you can get Play to generate a router that will declare all the controllers that it routes to as dependencies, allowing your controllers to be dependency injected themselves.

We recommend always using the injected routes generator, the static routes generator exists primarily as a tool to aid migration so that existing projects don't have to make all their controllers non static at once.

To enable the injected routes generator, add the following to your build settings in `build.sbt`:

```
routesGenerator := InjectedRoutesGenerator
```

When using the injected routes generator, prefixing the action with an `@` symbol takes on a special meaning, it means instead of the controller being injected directly, a `Provider` of the controller will be injected. This allows, for example, prototype controllers, as well as an option for breaking cyclic dependencies.

## Injected actions

If using the static routes generator, you can indicate that an action has an injected controller by prefixing the action with `@`, like so:

```
GET /some/path @controllers.Application.index
```

# Component lifecycle

The dependency injection system manages the lifecycle of injected components, creating them as needed and injecting them into other components. Here's how component lifecycle works:

- **New instances are created every time a component is needed.** If a component is used more than once, then, by default, multiple instances of the component will be created. If you only want a single instance of a component then you need to mark it as a [singleton](#).
- **Instances are created lazily when they are needed.** If a component is never used by another component, then it won't be created at all. This is usually what you want. For most components there's no point creating them until they're needed. However, in some cases you want components to be started up straight away or even if they're not used by another component. For example, you might want to send a message to a remote system or warm up a cache when the application starts. You can force a component to be created eagerly by using an [eager binding](#).
- **Instances are not automatically cleaned up,** beyond normal garbage collection. Components will be garbage collected when they're no longer referenced, but the framework won't do anything special to shut down the component, like calling a `close` method. However, Play provides a special type of component, called the [ApplicationLifecycle](#) which lets you register components to [shut down when the application stops](#).

## Singletons

Sometimes you may have a component that holds some state, such as a cache, or a connection to an external resource, or a component might be expensive to create. In these cases it may be important that there is only one instance of that component. This can be achieved using the [@Singleton](#) annotation:

```
import javax.inject._
```

```
@Singleton
class CurrentSharePrice {
 @volatile private var price = 0

 def set(p: Int) = price = p
 def get = price
}
```

## Stopping/cleaning up

Some components may need to be cleaned up when Play shuts down, for example, to stop thread pools. Play provides an [ApplicationLifecycle](#) component that can be used to register hooks to stop your component when Play shuts down:

```
import scala.concurrent.Future
import javax.inject._
import play.api.inject.ApplicationLifecycle

{@Singleton
class MessageQueueConnection @Inject()(lifecycle: ApplicationLifecycle) {
 val connection = connectToMessageQueue()
 lifecycle.addStopHook { () =>
 Future.successful(connection.stop())
 }

 ...
}
```

The `ApplicationLifecycle` will stop all components in reverse order from when they were created. This means any components that you depend on can still safely be used in your components stop hook, since because you depend on them, they must have been created before your component was, and therefore won't be stopped until after your component is stopped.

**Note:** It's very important to ensure that all components that register a stop hook are singletons. Any non singleton components that register stop hooks could potentially be a source of memory leaks, since a new stop hook will be registered each time the component is created.

# Providing custom bindings

It is considered good practice to define a trait for a component, and have other classes depend on that trait, rather than the implementation of the component. By doing that, you can inject different implementations, for example you inject a mock implementation when testing your application.

In this case, the DI system needs to know which implementation should be bound to that trait. The way we recommend that you declare this depends on whether you are writing a Play application as an end user of Play, or if you are writing library that other Play applications will consume.

## Play applications

We recommend that Play applications use whatever mechanism is provided by the DI framework that the application is using. Although Play does provide a binding API, this API is somewhat limited, and will not allow you to take full advantage of the power of the framework you're using.

Since Play provides support for Guice out of the box, the examples below show how to provide bindings for Guice.

### *Binding annotations*

The simplest way to bind an implementation to an interface is to use the

Guice [@ImplementedBy](#) annotation. For example:

```
import com.google.injectImplementedBy

@ImplementedBy(classOf[EnglishHello])
trait Hello {
 def sayHello(name: String): String
}

class EnglishHello extends Hello {
 def sayHello(name: String) = "Hello " + name
}
```

### *Programmatic bindings*

In some more complex situations, you may want to provide more complex bindings, such as when you have multiple implementations of the one trait, which are qualified by [@Named](#) annotations. In these cases, you can implement a custom Guice [Module](#):

```
import com.google.inject.AbstractModule
import com.google.inject.name.Names
```

```
class HelloModule extends AbstractModule {
 def configure() = {

 bind(classOf[Hello])
 .annotatedWith(Names.named("en"))
 .to(classOf[EnglishHello])

 bind(classOf[Hello])
 .annotatedWith(Names.named("de"))
 .to(classOf[GermanHello])
 }
}
```

To register this module with Play, append it's fully qualified class name to

the `play.modules.enabled` list in `application.conf`:

```
play.modules.enabled += "modules.HelloModule"
```

### *Configurable bindings*

Sometimes you might want to read the Play [Configuration](#) or use a [ClassLoader](#) when you configure Guice bindings. You can get access to these objects by adding them to your module's constructor.

In the example below, the `Hello` binding for each language is read from a configuration file. This allows new `Hello` bindings to be added by adding new settings in your `application.conf` file.

```
import com.google.inject.AbstractModule
import com.google.inject.name.Names
import play.api.{ Configuration, Environment }
```

```

class HelloModule(
 environment: Environment,
 configuration: Configuration) extends AbstractModule {
 def configure() = {
 // Expect configuration like:
 // hello.en = "myapp.EnglishHello"
 // hello.de = "myapp.GermanHello"
 val helloConfiguration: Configuration =
 configuration.getConfig("hello").getOrElse(Configuration.empty)
 val languages: Set[String] = helloConfiguration.subKeys
 // Iterate through all the languages and bind the
 // class associated with that language. Use Play's
 // ClassLoader to load the classes.
 for (l <- languages) {
 val bindingClassName: String = helloConfiguration.getString(l).get
 val bindingClass: Class[_ <: Hello] =
 environment.classLoader.loadClass(bindingClassName)
 .asSubclass(classOf[Hello])
 bind(classOf[Hello])
 .annotatedWith(Names.named(l))
 .to(bindingClass)
 }
 }
}

```

**Note:** In most cases, if you need to access Configuration when you create a component, you should inject the Configuration object into the component itself or into the component's Provider. Then you can read the Configuration when you create the component. You usually don't need to read Configuration when you create the bindings for the component.

### Eager bindings

In the code above, new EnglishHello and GermanHello objects will be created each time they are used. If you only want to create these objects once, perhaps because they're expensive to create, then you should use the @Singleton annotation as described above. If you want to create them once and also create them *eagerly* when the application starts up, rather than lazily when they are needed, then you can [Guice's eager singleton binding](#).

```

import com.google.inject.AbstractModule
import com.google.inject.name.Names

class HelloModule extends AbstractModule {
 def configure() = {

 bind(classOf[Hello])
 .annotatedWith(Names.named("en"))
 .to(classOf[EnglishHello]).asEagerSingleton

 bind(classOf[Hello])
 .annotatedWith(Names.named("de"))
 .to(classOf[GermanHello]).asEagerSingleton
 }
}

```

```
}
```

Eager singletons can be used to start up a service when an application starts. They are often combined with a [shutdown hook](#) so that the service can clean up its resources when the application stops.

## Play libraries

If you're implementing a library for Play, then you probably want it to be DI framework agnostic, so that your library will work out of the box regardless of which DI framework is being used in an application. For this reason, Play provides a lightweight binding API for providing bindings in a DI framework agnostic way.

To provide bindings, implement a [Module](#) to return a sequence of the bindings that you want to provide. The `Module` trait also provides a DSL for building bindings:

```
import play.api.inject._
```

```
class HelloModule extends Module {
 def bindings(environment: Environment,
 configuration: Configuration) = Seq(
 bind[Hello].qualifiedWith("en").to[EnglishHello],
 bind[Hello].qualifiedWith("de").to[GermanHello]
)
}
```

This module can be registered with Play automatically by appending it to the `play.modules.enabled` list in `reference.conf`:

```
play.modules.enabled += "com.example.HelloModule"
```

- The `Module.bindings` method takes a Play `Environment` and `Configuration`. You can access these if you want to [configure the bindings dynamically](#).
- Module bindings support [eager bindings](#). To declare an eager binding, add `.eagerly` at the end of your `Binding`.

In order to maximise cross framework compatibility, keep in mind the following things:

- Not all DI frameworks support just in time bindings. Make sure all components that your library provides are explicitly bound.
- Try to keep binding keys simple - different runtime DI frameworks have very different views on what a key is and how it should be unique or not.

## Excluding modules

If there is a module that you don't want to be loaded, you can exclude it by appending it to the `play.modules.disabled` property in `application.conf`:

```
play.modules.disabled += "play.api.db.evolutions.EvolutionsModule"
```

# Advanced: Extending the GuiceApplicationLoader

Play's runtime dependency injection is bootstrapped by the `GuiceApplicationLoader` class. This class loads all the modules, feeds the modules into Guice, then uses Guice to create the application. If you want to control how Guice initializes the application then you can extend the `GuiceApplicationLoader` class.

There are several methods you can override, but you'll usually want to override the `builder` method. This method reads the `ApplicationLoader.Context` and creates a `GuiceApplicationBuilder`. Below you can see the standard implementation for `builder`, which you can change in any way you like. You can find out how to use the `GuiceApplicationBuilder` in the section about [testing with Guice](#).

```
import play.api.ApplicationLoader
import play.api.Configuration
import play.api.inject._
import play.api.inject.guice._
```

```
class CustomApplicationLoader extends GuiceApplicationLoader() {
 override def builder(context: ApplicationLoader.Context): GuiceApplicationBuilder = {
 val extra = Configuration("a" -> 1)
 initialBuilder
 .in(context.environment)
 .loadConfig(extra ++ context.initialConfiguration)
 .overrides(overrides(context): _*)
 }
}
```

When you override the `ApplicationLoader` you need to tell Play. Add the following setting to your `application.conf`:

```
play.application.loader = "modules.CustomApplicationLoader"
```

You're not limited to using Guice for dependency injection. By overriding the `ApplicationLoader` you can take control of how the application is initialized. Find out more in the [next section](#).

Next: [Compile time dependency injection](#)

# Compile Time Dependency Injection

Out of the box, Play provides a mechanism for runtime dependency injection - that is, dependency injection where dependencies aren't wired until runtime. This approach has both advantages and disadvantages, the main advantages being around minimisation of boilerplate code, the main disadvantage being that the construction of the application is not validated at compile time.

An alternative approach that is popular in Scala development is to use compile time dependency injection. At its simplest, compile time DI can be achieved by manually constructing and wiring dependencies. Other more advanced techniques and tools exist, such as macro based autowiring tools, implicit auto wiring techniques, and various forms of the cake pattern. All of these can be easily implemented on top of constructors and manual wiring, so Play's support for compile time dependency injection is provided by providing public constructors and factory methods as API.

In addition to providing public constructors and factory methods, all of Play's out of the box modules provide some traits that implement a lightweight form of the cake pattern, for convenience. These are built on top of the public constructors, and are completely optional. In some applications, they will not be appropriate to use, but in many applications, they will be a very convenient mechanism to wiring the components provided by Play. These traits follow a naming convention of ending the trait name with `Components`, so for example, the default HikariCP based implementation of the DB API provides a trait called [HikariCPComponents](#).

In the examples below, we will show how to wire a Play application manually using the built in component helper traits. By reading the source code of these traits it should be trivial to adapt this to any compile time dependency injection technique you please.

---

## Current application

One aim of dependency injection is to eliminate global state, such as singletons. Play 2 was designed with an assumption of global state. Play 3 will hopefully remove this global state, however that is a major breaking task. In the meantime, Play will be a bit of a hybrid state, with some parts not using global state, and other parts using global state.

By using dependency injection throughout your application, you should be able to ensure though that your components can be tested in isolation, not requiring starting an entire application to run a single test.

---

## Application entry point

Every application that runs on the JVM needs an entry point that is loaded by reflection - even if your application starts itself, the main class is still loaded by reflection, and its main method is located and invoked using reflection.

In Play's dev mode, the JVM and HTTP server used by Play must be kept running between restarts of your application. To implement this, Play provides an [ApplicationLoader](#) trait that you can implement. The application loader is constructed and invoked every time the application is reloaded, to load the application.

This trait's load method takes as an argument the application loader [Context](#), which contains all the components required by a Play application that outlive the application itself and cannot be constructed by the application itself. A number of these components exist specifically for the purposes of providing functionality in dev mode, for example, the source mapper allows the Play error handlers to render the source code of the place that an exception was thrown.

The simplest implementation of this can be provided by extending the [PlayBuiltInComponentsFromContext](#) abstract class. This class takes the context, and provides all the built in components, based on that context. The only thing you need to provide is a router for Play to route requests to. Below is the simplest application that can be created in this way, using a null router:

```
import play.api._
import play.api.ApplicationLoader.Context
import play.api.routing.Router

class MyApplicationLoader extends ApplicationLoader {
 def load(context: Context) = {
 new MyComponents(context).application
 }
}

class MyComponents(context: Context) extends BuiltInComponentsFromContext(context) {
 lazy val router = Router.empty
}
```

To configure Play to use this application loader, configure the `play.application.loader` property to point to the fully qualified class name in the `application.conf` file:

```
play.application.loader=MyApplicationLoader
```

## Providing a router

By default Play will generate a static router that requires all of your actions to be objects. Play however also supports generating a router than can be dependency injected, this can be enabled by adding the following configuration to your `build.sbt`:

```
routesGenerator := InjectedRoutesGenerator
```

When you do this, Play will generate a router with a constructor that accepts each of the controllers and included routers from your routes file, in the order they appear in your routes file. The routers constructor will also, as its first argument, accept an [HttpErrorHandler](#), which is used to handle parameter binding errors. The primary constructor will also accept a

prefix String as the last argument, but an overloaded constructor that defaults this to `" / "` will also be provided.

The following routes:

```
GET / controllers.Application.index
GET /foo controllers.Application.foo
-> /bar bar.Routes
GET /assets/*file controllers.Assets.at(path = "/public", file)
```

Will produce a router with the following constructor signatures:

```
class Routes(
 override val errorHandler: play.api.http.HttpErrorHandler,
 Application_0: controllers.Application,
 bar_Routes_0: bar.Routes,
 Assets_1: controllers.Assets,
 val prefix: String
) extends GeneratedRouter {

 def this(
 errorHandler: play.api.http.HttpErrorHandler,
 Application_0: controllers.Application,
 bar_Routes_0: bar.Routes,
 Assets_1: controllers.Assets
) = this(Application_0, bar_Routes_0, Assets_1, "/")

 ...
}
```

Note that the naming of the parameters is intentionally not well defined (and in fact the index that is appended to them is random, depending on hash map ordering), so you should not depend on the names of these parameters.

To use this router in an actual application:

```
import play.api._
import play.api.ApplicationLoader.Context
import router.Routes

class MyApplicationLoader extends ApplicationLoader {
 def load(context: Context) = {
 new MyComponents(context).application
 }
}

class MyComponents(context: Context) extends BuiltInComponentsFromContext(context) {

 lazy val router = new Routes(errorHandler, applicationController, barRoutes, assets)

 lazy val barRoutes = new bar.Routes(errorHandler)
 lazy val applicationController = new controllers.Application()
```

```
lazy val assets = new controllers.Assets(httpErrorHandler)
}
```

# Using other components

As described before, Play provides a number of helper traits for wiring in other components. For example, if you wanted to use the messages module, you can mix in [I18nComponents](#) into your components cake, like so:

```
import play.api.i18n._
```

```
class MyComponents(context: Context) extends BuiltInComponentsFromContext(context)
 with I18nComponents {
 lazy val router = Router.empty

 lazy val myComponent = new MyComponent(messagesApi)
}

class MyComponent(messages: MessagesApi) {
 // ...
}
```

Next: [Advanced routing](#)

# String Interpolating Routing DSL

Play provides a DSL for defining embedded routers called the *String Interpolating Routing DSL*, or sird for short. This DSL has many uses, including embedding a light weight Play server, providing custom or more advanced routing capabilities to a regular Play application, and mocking REST services for testing.

Sird is based on a string interpolated extractor object. Just as Scala supports interpolating parameters into strings for building strings (and any object for that matter), such as `s"Hello $to"`, the same mechanism can also be used to extract parameters out of strings, for example in case statements.

The DSL lives in the `play.api.routing.sird` package. Typically, you will want to import this package, as well as a few other packages:

```
import play.api.mvc._
import play.api.routing._
import play.api.routing.sird._
```

A simple example of its use is:

```
val router = Router.from {
```

```

case GET(p"/hello/$to") => Action {
 Results.Ok(s"Hello $to")
}
}

```

In this case, the `$to` parameter in the interpolated path pattern will extract a single path segment for use in the action. The `GET` extractor extracts requests with the `GET` method. It takes a `RequestHeader` and extracts the same `RequestHeader` parameter, it's only used as a convenient filter. Other method extractors, including `POST`, `PUT` and `DELETE` are also supported.

Like Play's compiled router, sird supports matching multi path segment parameters, this is done by postfixing the parameter with `*`:

```

val router = Router.from {
 case GET(p"/assets/$file*") =>
 Assets.versioned(path = "/public", file = file)
}

```

Regular expressions are also supported, by postfixing the parameter with a regular expression in angled brackets:

```

val router = Router.from {
 case GET(p"/items/$id<[0-9]+>") => Action {
 Results.Ok(s"Item $id")
 }
}

```

Query parameters can also be extracted, using the `?` operator to do further extractions on the request, and using the `q` extractor:

```

val router = Router.from {
 case GET(p"/search" ? q("query=$query")) => Action {
 Results.Ok(s"Searching for $query")
 }
}

```

While `q` extracts a required query parameter as a `String`, `q_?` or `q_o` if using Scala 2.10 extracts an optional query parameter as `Option[String]`:

```

val router = Router.from {
 case GET(p"/items" ? q_o("page=$page")) => Action {
 val thisPage = page.getOrElse("1")
 Results.Ok(s"Showing page $thisPage")
 }
}

```

Likewise, `q_*` or `q_s` can be used to extract a sequence of multi valued query parameters:

```

val router = Router.from {
 case GET(p"/items" ? q_s("tag=$tags")) => Action {
 val allTags = tags.mkString(", ")
 Results.Ok(s"Showing items tagged: $allTags")
 }
}

```

Multiple query parameters can be extracted using the `&` operator:

```

val router = Router.from {
 case GET(p"/items" ? q_o("page=$page"))

```

```

& q_o"per_page=$perPage") => Action {
 val thisPage = page.getOrElse("1")
 val pageLength = perPage.getOrElse("10")

 Results.Ok(s"Showing page $thisPage of length $pageLength")
}
}

```

Since sird is just a regular extractor object (built by string interpolation), it can be combined with any other extractor object, including extracting its sub parameters even further. Sird provides some useful extractors for some of the most common types out of the box, namely `int`, `long`, `float`, `double` and `bool`:

```

val router = Router.from {
 case GET(p"/items/${int(id)}") => Action {
 Results.Ok(s"Item $id")
 }
}

```

In the above, `id` is of type `Int`. If the `int` extractor failed to match, then of course, the whole pattern will fail to match.

Similarly, the same extractors can be used with query string parameters, including multi value and optional query parameters. In the case of optional or multi value query parameters, the match will fail if any of the values present can't be bound to the type, but no parameters present doesn't cause the match to fail:

```

val router = Router.from {
 case GET(p"/items" ? q_o"page=${int(page)})" => Action {
 val thePage = page.getOrElse(1)
 Results.Ok(s"Items page $thePage")
 }
}

```

To further the point that these are just regular extractor objects, you can see here that you can use all other features of a `case` statement, including `@` syntax and if statements:

```

val router = Router.from {
 case rh @ GET(p"/items/${idString @ int(id)}" ?
 q"price=${int(price)}") ?
 if price > 200 =>
 Action {
 Results.Ok(s"Expensive item $id")
 }
}

```

**Next:** [Javascript routing](#)

# Javascript Routing

The play router is able to generate Javascript code to handle routing from Javascript running client side back to your application. The Javascript router aids in refactoring your application. If you change the structure of your URLs or parameter names your Javascript gets automatically updated to use that new structure.

---

# Generating a Javascript router

The first step to using Play's Javascript router is to generate it. The router will only expose the routes that you explicitly declare thus minimising the size of the Javascript code.

There are two ways to generate a Javascript router. One is to embed the router in the HTML page using template directives. The other is to generate Javascript resources in an action that can be downloaded, cached and shared between pages.

## Embedded router

An embedded router can be generated using the `@javascriptRouter` directive inside a Scala template. This is typically done inside the main decorating template.

```
@helper.javascriptRouter("jsRoutes")(
 routes.javascript.Users.list,
 routes.javascript.Users.get
)
```

The first parameter is the name of the global variable that the router will be placed in. The second parameter is the list of Javascript routes that should be included in this router. In order to use this function, your template must have an implicit `RequestHeader` in scope. For example this can be made available by adding `(implicit req: RequestHeader)` to the end of your parameter declarations.

## Router resource

A router resource can be generated by creating an action that invokes the router generator. It has a similar syntax to embedding the router in a template:

```
def javascriptRoutes = Action { implicit request =>
 Ok(
 JavaScriptReverseRouter("jsRoutes")(
 routes.javascript.Users.list,
 routes.javascript.Users.get
)
 .as("text/javascript")
)
```

Then, add the corresponding route:

```
GET /javascriptRoutes controllers.Application.javascriptRoutes
```

Having implemented this action, and adding it to your routes file, you can then include it as a resource in your templates:

```
<script type="text/javascript" src="@routes.Application.javascriptRoutes"></script>
```

# Using the router

Using jQuery as an example, making a call is as simple as:

```
$.ajax(jsRoutes.controllers.Users.get(someId))
 .done(/*...*/)
 .fail(/*...*/);
```

The router also makes a few other properties available including the `url` and the `type`(the HTTP method). For example the above call to jQuery's ajax function can also be made like:

```
var r = jsRoutes.controllers.Users.get(someId);
$.ajax({url: r.url, type: r.type, success: /*...*/, error: /*...*/});
```

The above approach is required where other properties need setting such as success, error, context etc.

The `absoluteURL` and the `webSocketURL` are methods (not properties) which return the complete url string. A Websocket connection can be made like:

```
var r = jsRoutes.controllers.Users.list();
var ws = new WebSocket(r.webSocketURL());
ws.onmessage = function(msg) {
 /*...*/
};
```

## jQuery ajax method support

**Note:** Built-in support for jQuery's ajax function will be removed in a future release. This section on the built-in support is provided for reference purposes only. Please do not use the router's ajax function in new code and consider upgrading existing code as soon as possible. The previous section on using the router documents how jQuery should be used.

If jQuery isn't your thing, or if you'd like to decorate the jQuery ajax method in some way, you can provide a function to the router to use to perform ajax queries. This function must accept the object that is passed to the `ajax` router method, and should expect the router to have set the `type` and `url` properties on it to the appropriate method and url for the router request.

To define this function, in your action pass the `ajaxMethod` method parameter, eg:

```
Routes.javascriptRouter("jsRoutes", Some("myAjaxFunction") ...
```

Next: [Extending Play](#)

# Writing Plugins

**Note:** Plugins are deprecated. Instead, use [Modules](#).

In the context of the Play runtime, a plugin is a class that is able to plug into the Play lifecycle, and also allows sharing components in a non static way in your application.

Not every library that adds functionality to Play is or needs to be a plugin in this context - a library that provides a custom filter for example does not need to be a plugin.

Similarly, plugins don't necessarily imply that they are reusable between applications, it is often very useful to implement a plugin locally within an application, in order to hook into the Play lifecycle and share components in your code.

## Implementing plugins

Implementing a plugin requires two steps. The first is to implement the `play.api.Plugin` interface:

package plugins

```
import play.api.{Plugin, Application}

class MyPlugin extends Plugin {
 val myComponent = new MyComponent()

 override def onStart() = {
 myComponent.start()
 }

 override def onStop() = {
 myComponent.stop()
 }

 override def enabled = true
}
```

The next step is to register this with Play. This can be done by creating a file called `play.plugins` and placing it in the root of the classloader. In a typical Play app, this means putting it in the `conf` folder:

2000:plugins.MyPlugin

Each line in the `play.plugins` file contains a number followed by the fully qualified name of the plugin to load. The number is used to control lifecycle ordering, lower numbers will be started first and stopped last. Multiple plugins can be declared in the one file, and any lines started with `#` are treated as comments.

Choosing the right number for ordering for a plugin is important, it needs to fit in appropriate according to what other plugins it depends on. The plugins that Play uses use the following ordering numbers:

- 100 - Utilities that have no dependencies, such as the messages plugin
- 200 - Database connection pools
- 300-500 - Plugins that depend on the database, such as JPA, Ebean and evolutions
- 600 - The Play cache plugin
- 700 - The WS plugin
- 1000 - The Akka plugin
- 10000 - The Global plugin, which invokes the `Global.onStart` and `Global.onStop` methods. This plugin is intended to execute last.

## Accessing plugins

Plugins can be accessed via the `plugin` method on `play.api.Application`:

```
import play.api.Play
import play.api.Play.current

val myComponent = Play.application.plugin[MyPlugin]
 .getOrElse(throw new RuntimeException("MyPlugin not loaded"))
 .myComponent
```

## Actor example

A common use case for using plugins is to create and share actors around the application. This can be done by implementing an actors plugin:

```
package actors

import play.api._
import play.api.libs.concurrent.Akka
import akka.actor._
import javax.inject.Inject

class Actors @Inject()(implicit app: Application) extends Plugin {
 lazy val myActor = Akka.system.actorOf(MyActor.props, "my-actor")
}

object Actors {
 def myActor: ActorRef = Play.current.plugin[Actors]
 .getOrElse(throw new RuntimeException("Actors plugin not loaded"))
 .myActor
}
```

Note the `Actors` companion object methods that allow easy access to the `ActorRef` for each actor, instead of code having to use the plugins API directly.

The plugin can then be registered in `play.plugins`:

1100:actors.Actors

The reason 1100 was chosen for the ordering was because this plugin depends on the Akka plugin, and so must start after that.

Next: [Embedding Play](#)

# Embedding a Play server in your application

While Play apps are most commonly used as their own container, you can also embed a Play server into your own existing application. This can be used in conjunction with the Twirl template compiler and Play routes compiler, but these are of course not necessary, a common use case for embedding a Play application will be because you only have a few very simple routes.

The simplest way to start an embedded Play server is to use the `NettyServer` factory methods. If all you need to do is provide some straightforward routes, you may decide to use the [String Interpolating Routing DSL](#) in combination with the `fromRouter` method:

```
import play.core.server._
import play.api.routing.sird._
import play.api.mvc._

val server = NettyServer.fromRouter() {
 case GET(p"/hello/$to") => Action {
 Results.Ok(s"Hello $to")
 }
}
```

By default, this will start a server on port 9000 in prod mode. You can configure the server by passing in a `ServerConfig`:

```
import play.core.server._
import play.api.routing.sird._
import play.api.mvc._

val server = NettyServer.fromRouter(ServerConfig(
 port = Some(19000),
 address = "127.0.0.1"
) {
 case GET(p"/hello/$to") => Action {
 Results.Ok(s"Hello $to")
 }
}
```

You may want to customise some of the components that Play provides, for example, the HTTP error handler. A simple way of doing this is by using Play's components traits,

the `NettyServerComponents` trait is provided for this purpose, and can be conveniently combined with `BuiltInComponents` to build the application that it requires:

```
import play.core.server._
import play.api.routing.Router
import play.api.routing.sird._
import play.api.mvc._
import play.api.BuiltInComponents
import play.api.http.DefaultHttpErrorHandler
import scala.concurrent.Future

val components = new NettyServerComponents with BuiltInComponents {

 lazy val router = Router.from {
 case GET(p"/hello/$to") => Action {
 Results.Ok(s"Hello $to")
 }
 }

 override lazy val httpErrorHandler = new DefaultHttpErrorHandler(environment,
 configuration, sourceMapper, Some(router)) {

 override protected def onNotFound(request: RequestHeader, message: String) = {
 Future.successful(Results.NotFound("Nothing was found!"))
 }
 }
}
val server = components.server
```

In this case, the server configuration can be overridden by overriding the `serverConfig` property.

To stop the server once you've started it, simply call the `stop` method:

```
server.stop()
```

**Note:** Play requires an application secret to be configured in order to start. This can be configured by providing an `application.conf` file in your application, or using the `play.crypto.secret` system property.

**Next:** [Play for Java developers](#)

# The Build System

The Play build system uses `sbt`, a high-performance integrated build for Scala and Java projects. Using `sbt` as our build tool brings certain requirements to play which are explained on this page.

## Understanding sbt

sbt functions quite differently to the way many traditional build tasks. Fundamentally, sbt is a task engine. Your build is represented as a tree of task dependencies that need to be executed, for example, the `compile` task depends on the `sources` task, which depends on the `sourceDirectories` task and the `sourceGenerators` task, and so on.

sbt breaks typical build executions up into very fine grained tasks, and any task at any point in the tree can be arbitrarily redefined in your build. This makes sbt very powerful, but also requires a shift in thinking if you've come from other build tools that break your build up into very coarsely grained tasks.

The documentation here describes Play's usage of sbt at a very high level. As you start to use sbt more in your project, it is recommended that you follow the [sbt tutorial](#) to get an understanding for how sbt fits together. Another resource that many people have found useful is [this series of blog posts](#).

---

# Play application directory structure

Most people get started with Play using the `activator new` command which produces a directory structure like this:

- `/`: The root folder of your application
- `/README`: A text file describing your application that will get deployed with it.
- `/app`: Where your application code will be stored.
- `/build.sbt`: The `sbt` settings that describe building your application.
- `/conf`: Configuration files for your application
- `/project`: Further build description information
- `/public`: Where static, public assets for your application are stored.
- `/test`: Where your application's test code will be stored.

For now, we are going to concern ourselves with the `/build.sbt` file and the `/project` directory.

---

## The `/build.sbt` file.

When you use the `activator new foo` command, the build description file, `/build.sbt`, will be generated like this:

```
name := "foo"
```

```
version := "1.0-SNAPSHOT"
```

```
libraryDependencies ++= Seq(
 jdbc,
 anorm,
```

```
cache
)

lazy val root = (project in file(".")).enablePlugins(PlayScala)
```

The `name` line defines the name of your application and it will be the same as the name of your application's root directory, `/`, which is derived from the argument that you gave to the `activator new` command.

The `version` line provides the version of your application which is used as part of the name for the artifacts your build will produce.

The `libraryDependencies` line specifies the libraries that your application depends on. More on this below.

You should use the `PlayJava` or `PlayScala` plugin to configure sbt for Java or Scala respectively.

## Using scala for building

Activator is also able to construct the build requirements from scala files inside your project's `project` folder. The recommended practice is to use `build.sbt` but there are times when using scala directly is required. If you find yourself, perhaps because you're migrating an older project, then here are a few useful imports:

```
import sbt._
import Keys._
import play.Play.autoImport._
import PlayKeys._
```

The line indicating `autoImport` is the correct means of importing an sbt plugin's automatically declared properties. Along the same lines, if you're importing an sbt-web plugin then you might well:

```
import com.typesafe.sbt.less.autoImport._
import LessKeys._
```

# The `/project` directory

Everything related to building your project is kept in the `/project` directory underneath your application directory. This is an `sbt` requirement. Inside that directory, there are two files:

- `/project/build.properties`: This is a marker file that declares the sbt version used.
- `/project/plugins.sbt`: SBT plugins used by the project build including Play itself.

# Play plugin for sbt (`/project/plugins.sbt`)

The Play console and all of its development features like live reloading are implemented via an sbt plugin. It is registered in the `/project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % playVersion) // where version is the current Play version,
i.e. "2.4.x"
```

Note that `build.properties` and `plugins.sbt` must be manually updated when you are changing the play version.

Next: [About sbt settings](#)

# About SBT Settings

## About sbt settings

The `build.sbt` file defines settings for your project. You can also define your own custom settings for your project, as described in the [sbt documentation](#). In particular, it helps to be familiar with the `settings` in sbt.

To set a basic setting, use the `:=` operator:

```
confDirectory := "myConfFolder"
```

## Default settings for Java applications

Play defines a default set of settings suitable for Java-based applications. To enable them add the `PlayJava` plugin via your project's `enablePlugins` method. These settings mostly define the default imports for generated templates e.g. importing `java.lang.*` so types like `Long` are the Java ones by default instead of the Scala ones. `play.Project.playJavaSettings` also imports `java.util.*` so that the default collection library will be the Java one.

## Default settings for Scala applications

Play defines a default set of settings suitable for Scala-based applications. To enable them add the `PlayScala` plugin via your project's `enablePlugins` method. These default settings define the default imports for generated templates (such as internationalized messages, and core APIs).

Next: [Manage application dependencies](#)

# Managing library dependencies

## Unmanaged dependencies

Most people end up using managed dependencies - which allows for fine-grained control, but unmanaged dependencies can be simpler when starting out.

Unmanaged dependencies work like this: create a `lib/` directory in the root of your project and then add jar files to that directory. They will automatically be added to the application classpath. There's not much else to it!

There's nothing to add to `build.sbt` to use unmanaged dependencies, although you could change a configuration key if you'd like to use a directory different to `lib`.

## Managed dependencies

Play uses Apache Ivy (via sbt) to implement managed dependencies, so if you're familiar with Maven or Ivy, you won't have much trouble.

Most of the time you can simply list your dependencies in the `build.sbt` file.

Declaring a dependency looks like this (defining `group`, `artifact` and `revision`):

```
libraryDependencies += "org.apache.derby" % "derby" % "10.11.1.1"
```

or like this, with an optional `configuration`:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.11.1.1" % "test"
```

Multiple dependencies can be added either by multiple declarations like the above, or you can provide a Scala sequence:

```
libraryDependencies ++= Seq(
 "org.apache.derby" % "derby" % "10.11.1.1",
 "org.hibernate" % "hibernate-entitymanager" % "4.3.9.Final"
)
```

Of course, sbt (via Ivy) has to know where to download the module. If your module is in one of the default repositories sbt comes with then this will just work.

## Getting the right Scala version with

If you use `groupID %% artifactID % revision` instead of `groupID % artifactID % revision` (the difference is the double `%%` after the `groupID`), sbt will add your project's Scala version to the artifact name. This is just a shortcut.

You could write this without the `%%`:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.9.1" % "0.3"
```

Assuming the `scalaVersion` for your build is `2.9.1`, the following is identical:

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

## Resolvers

sbt uses the standard Maven2 repository and the Typesafe Releases (<https://repo.typesafe.com/typesafe/releases>) repositories by default. If your dependency isn't on one of the default repositories, you'll have to add a resolver to help Ivy find it.

Use the `resolvers` setting key to add your own resolver.

```
resolvers += name at location
```

For example:

```
resolvers += "sonatype snapshots" at "https://oss.sonatype.org/content/repositories/snapshots/"
```

sbt can search your local Maven repository if you add it as a repository:

```
resolvers += (
 "Local Maven Repository" at "file:///"+Path.userHome.absolutePath+"/.m2/repository"
)
```

**Next:** [Working with sub-projects](#)

# Working with sub-projects

A complex project is not necessarily composed of a single Play application. You may want to split a large project into several smaller applications, or even extract some logic into a standard Java or Scala library that has nothing to do with a Play application.

It will be helpful to read the [SBT documentation on multi-project builds](#). Sub-projects do not have their own build file, but share the parent project's build file.

## Adding a simple library sub-project

You can make your application depend on a simple library project. Just add another sbt project definition in your `build.sbt` file:

```

name := "my-first-application"
version := "1.0"

lazy val myFirstApplication = (project in file("."))

 .enablePlugins(PlayScala)

 .aggregate(myLibrary)

 .dependsOn(myLibrary)

```

lazy val myLibrary = project

The lowercased `project` on the last line is a Scala Macro which will use the name of the `val` it is being assigned to in order to determine the project's name and folder.

The `myFirstApplication` project declares the base project. If you don't have any sub projects, this is already implied, however when declaring sub projects, it's usually required to declare it so that you can ensure that it aggregates (that is, runs things like `compile/test` etc on the sub projects when run in the base project) and depends on (that is, adds the sub projects to the main projects classpath) the sub projects.

The above example defines a sub-project in the application's `myLibrary` folder. This sub-project is a standard sbt project, using the default layout:

```

myProject
└ build.sbt
└ app
└ conf
└ public
└ myLibrary
 └ build.sbt
 └ src
 └ main
 └ java
 └ scala

```

`myLibrary` has its own `build.sbt` file, this is where it can declare its own settings, dependencies etc.

When you have a sub-project enabled in your build, you can focus on this project and compile, test or run it individually. Just use the `projects` command in the Play console prompt to display all projects:

```

[my-first-application] $ projects
[info] In file:/Volumes/Data/gbo/myFirstApp/
[info] * my-first-application
[info] my-library

```

The default project is the one whose variable name comes first alphabetically. You may make your main project by making its variable name `aaaMain`. To change the current project use the `project` command:

```

[my-first-application] $ project my-library
[info] Set current project to my-library
>

```

When you run your Play application in dev mode, the dependent projects are automatically recompiled, and if something cannot compile you will see the result in your browser:

A screenshot of a web browser window titled "Compilation error". The address bar shows "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below this, the error message "not found: type Strign" is displayed. Underneath, a code editor shows a Scala file named "Utils.scala" with line numbers 1 through 7. Line 5 contains the error: "def uppercase(s: Strign) = s.toUpperCase".

```
Compilation error
localhost:9000

Compilation error

not found: type Strign

In /Volumes/Data/gbo/myFirstApp/modules/myLibrary/src/main/scala/Utils.scala at line 5

1 package utils
2
3 object Helper {
4
5 def uppercase(s: Strign) = s.toUpperCase
6
7 }
```

## Sharing common variables and code

If you want your sub projects and root projects to share some common settings or code, then these can be placed in a Scala file in the `project` directory of the root project. For example, in `project/Common.scala` you might have:

```
import sbt._
import Keys._

object Common {
 val settings: Seq[Setting[_]] = Seq(
```

```
organization := "com.example",
version := "1.2.3-SNAPSHOT"
)

val fooDependency = "com.foo" %% "foo" % "2.4"
}
```

Then in each of your `build.sbt` files, you can reference anything declared in the file:  
name := "my-sub-module"

`Common.settings`

```
libraryDependencies += Common.fooDependency
```

# Splitting your web application into several parts

As a Play application is just a standard sbt project with a default configuration, it can depend on another Play application. You can make any sub module a Play application by adding the `PlayJava` or `PlayScala` plugins, depending on whether your project is a Java or Scala project, in its corresponding `build.sbt` file.

**Note:** In order to avoid naming collision, make sure your controllers, including the Assets controller in your subprojects are using a different name space than the main project

# Splitting the route file

It's also possible to split the route file into smaller pieces. This is a very handy feature if you want to create a robust, reusable multi-module play application

Consider the following build configuration

```
build.sbt:
name := "myproject"

lazy val admin = (project in file("modules/admin")).enablePlugins(PlayScala)

lazy val main = (project in file("."))
 .enablePlugins(PlayScala).dependsOn(admin).aggregate(admin)
modules/admin/build.sbt
name := "myadmin"

libraryDependencies ++= Seq(
 "mysql" % "mysql-connector-java" % "5.1.35",
 jdbc,
 anorm
)
```

## Project structure

```
build.sbt
app
 └ controllers
 └ models
 └ views
conf
 └ application.conf
 └ routes
modules
 └ admin
 └ build.sbt
 └ conf
 └ admin.routes
 └ app
 └ controllers
 └ models
 └ views
project
 └ build.properties
 └ plugins.sbt
```

**Note:** Configuration and route file names must be unique in the whole project structure. Particularly, there must be only one `application.conf` file and only one `routes` file. To define additional routes or configuration in sub-projects, use sub-project-specific names. For instance, the route file in `admin` is called `admin.routes`. To use a specific set of settings in development mode for a sub project, it would be even better to put these settings into the build file, e.g. `Keys.devSettings += ("play.http.router", "admin.Routes")`.

```
conf/routes:
```

```
GET /index controllers.Application.index()
```

```
-> /admin admin.Routes
```

```
GET /assets/*file controllers.Assets.at(path="/public", file)
```

```
modules/admin/conf/admin.routes:
```

```
GET /index controllers.admin.Application.index()
```

```
GET /assets/*file controllers.admin.Assets.at(path="/public/lib/myadmin", file)
```

**Note:** Resources are served from a unique classloader, and thus resource path must be relative from project classpath root.

Subprojects resources are generated in `target/web/public/main/lib/{module-name}`, so the resources are accessible from `/public/lib/{module-name}` when using `play.api.Application#resources(uri)` method, which is what the `Assets.at` method does.

**Assets and controller classes should be all defined in the `controllers.admin` package**

```
modules/admin/controllers/Assets.scala:
package controllers.admin
```

```

import play.api.http.LazyHttpErrorHandler
object Assets extends controllers.AssetsBuilder(LazyHttpErrorHandler)

Note: Java users can do something very similar i.e.:
// Assets.java
package controllers.admin;
import play.api.mvc.*;

public class Assets {
 public static Action<AnyContent> at(String path, String file) {
 return controllers.Assets.at(path, file);
 }
}

```

and a controller:

```

modules/admin/controllers/Application.scala:
package controllers.admin

import play.api._
import play.api.mvc._
import views.html._

object Application extends Controller {

 def index = Action { implicit request =>
 Ok("admin")
 }
}

```

Reverse routing in `admin`

in case of a regular controller call:

```

controllers.admin.routes.Application.index
and for Assets:
controllers.admin.routes.Assets.at("...")

```

Through the browser

`http://localhost:9000/index`

triggers

`controllers.Application.index`

and

`http://localhost:9000/admin/index`

triggers

`controllers.admin.Application.index`

# Play enhancer

The [Play enhancer](#) is an sbt plugin that generates getters and setters for Java beans, and rewrites the code that accesses those fields to use the getters and setters.

## Motivation

One common criticism of Java is that simple things require a lot of boilerplate code. One of the biggest examples of this is encapsulating fields - it is considered good practice to encapsulate the access and mutation of fields in methods, as this allows future changes such as validation and generation of the data. In Java, this means making all your fields private, and then writing getters and setters for each field, a typical overhead of 6 lines of code per field.

Furthermore, many libraries, particularly libraries that use reflection to access properties of objects such as ORMs, require classes to be implemented in this way, with getters and setters for each field.

The Play enhancer provides a convenient alternative to manually implementing getters and setters. It implements some post processing on the compiled byte code for your classes, this is commonly referred to as byte code enhancement. For every public field in your classes, Play will automatically generate a getter and setter, and then will rewrite the code that uses these fields to use the getters and setters instead.

## Drawbacks

Using byte code enhancement to generating getters and setters is not without its drawbacks however. Here are a few:

- Byte code enhancement is opaque, you can't see the getters and setters that are generated, so when things go wrong, it can be hard to debug and understand what is happening. Byte code enhancement is often described as being "magic" for this reason.
- Byte code enhancement can interfere with the operation of some tooling, such as IDEs, as they will be unaware of the eventual byte code that gets used. This can cause problems such as tests failing when run in an IDE because they depend on byte code enhancement, but the IDE isn't running the byte code enhancer when it compiles your source files.
- Existing Java developers that are new to your codebase will not expect getters and setters to be generated, this can cause confusion.

Whether you use the Play enhancer or not in your projects is up to you, if you do decide to use it the most important thing is that you understand what the enhancer does, and what the drawbacks may be.

---

# Setting up

To enable the byte code enhancer, simply add the following line to your `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-play-enhancer" % "1.1.0")
```

The Play enhancer should be enabled for all your projects. If you want to disable the Play enhancer for a particular project, you can do that like so in your `build.sbt` file:

```
lazy val nonEnhancedProject = (project in file("non-enhanced"))
 .disablePlugins(PlayEnhancer)
```

In some situations, it may not be possible to disable the enhancer plugin, an example of this is using Play's ebean plugin, which requires the enhancer to ensure that getters and setters are generated before it does its byte code enhancement. If you don't want to generate getters and setters in that case, you can use the `playEnhancerEnabled` setting:

```
playEnhancerEnabled := false
```

# Operation

The enhancer looks for all fields on Java classes that:

- are public
- are non static
- are non final

For each of those fields, it will generate a getter and a setter if they don't already exist. If you wish to provide a custom getter or setter for a field, this can be done by just writing it, the Play enhancer will simply skip the generation of the getter or setter if it already exists.

---

# Configuration

If you want to control exactly which files get byte code enhanced, this can be done by configuring the `sources` task scoped to

the `playEnhancerGenerateAccessors` and `playEnhancerRewriteAccessors` tasks. For example, to only enhance the java sources in the models package, you might do this:

```
sources in (Compile, playEnhancerGenerateAccessors) := {
 ((javaSource in Compile).value / "models" ** "*.java").get
}
```

Next: [Aggregating reverse routers](#)

# Aggregating reverse routers

In some situations you want to share reverse routers between sub projects that are not dependent on each other.

For example, you might have a `web` sub project, and an `api` sub project. These sub projects may have no dependence on each other, except that the `web` project wants to render links to the `api` project (for making AJAX calls), while the `api` project wants to render links to the `web` (rendering the web link for a resource in JSON). In this situation, it would be convenient to use the reverse router, but since these projects don't depend on each other, you can't.

Play's routes compiler offers a feature that allows a common dependency to generate the reverse routers for projects that depend on it so that the reverse routers can be shared between those projects. This is configured using the `aggregateReverseRoutes` sbt configuration item, like this:

```
lazy val common: Project = (project in file("common"))
 .enablePlugins(PlayScala)
 .settings(
 aggregateReverseRoutes := Seq(api, web)
)

lazy val api = (project in file("api"))
 .enablePlugins(PlayScala)
 .dependsOn(common)

lazy val web = (project in file("web"))
 .enablePlugins(PlayScala)
 .dependsOn(common)
```

In this setup, the reverse routers for `api` and `web` will be generated as part of the `common` project. Meanwhile, the forwards routers for `api` and `web` will still generate forwards routers, but not reverse routers, because their reverse routers have already been generated in the `common` project which they depend on, so they don't need to generate them.

Note that the `common` project has a type of `Project` explicitly declared. This is because there is a recursive reference between it and the `api` and `web` projects, through the `dependsOn` method and `aggregateReverseRoutes` setting, so the Scala type checker needs an explicit type somewhere in the chain of recursion.

Next: [Improving Compilation Times](#)

# Improving Compilation Times

---

Compilation speed can be improved by following some guidelines that are also good engineering practice:

## Use subprojects/modularize

This is something like bulkheads for incremental compilation in addition to the other benefits of modularization. It minimizes the size of cycles, makes inter-dependencies explicit, and allows you to work with a subset of the code when desired. It also allows sbt to compile independent modules in parallel.

---

## Annotate return types of public methods

This makes compilation faster as it reduces the need for type inference and for accuracy helps address corner cases in incremental compilation arising from inference across source file boundaries.

---

## Avoid large cycles between source files

Cycles tend to result in larger recompilations and/or more steps. In sbt 0.13.0+ (Play 2.2+), this is less of a problem.

---

## Minimize inheritance

A public API change in a source file typically requires recompiling all descendants.

# SBT Cookbook

## Hook actions around `play run`

You can apply actions around the `play run` command by extending `PlayRunHook`.

This trait define the following methods:

- `beforeStarted(): Unit`
- `afterStarted(addr: InetSocketAddress): Unit`
- `afterStopped(): Unit`

`beforeStarted` method is called before the play application is started, but after all “before run” tasks have been completed.

`afterStarted` method is called after the play application has been started.

`afterStopped` method is called after the play process has been stopped.

**Note:** The following example illustrate how you can start and stop a command with play run hook.

In the near future `sbt-web` will provide a better way to integrate Grunt with an SBT build.

Now let's say you want to build a Web application with `grunt` before the application is started.

First, you need to create a Scala object in the `project/` directory to extend `PlayRunHook`.

Let's name it `Grunt.scala`:

```
import play.PlayRunHook
import sbt._
```

```
object Grunt {
 def apply(base: File): PlayRunHook = {

 object GruntProcess extends PlayRunHook {

 override def beforeStarted(): Unit = {
 Process("grunt dist", base).run
 }
 }

 GruntProcess
 }
}
```

Then in the `build.sbt` file you need to register this hook:

```
import Grunt._
import play.PlayImport.PlayKeys.playRunHooks
```

```
playRunHooks <+= baseDirectory.map(base => Grunt(base))
```

This will execute the `grunt dist` command in `baseDirectory` before the application is started.

Now we want to execute `grunt watch` command to observe changes and rebuild the Web application when that happen:

```
import play.PlayRunHook
```

```
import sbt._
```

```
import java.net.InetSocketAddress
```

```
object Grunt {
```

```
 def apply(base: File): PlayRunHook = {
```

```
 object GruntProcess extends PlayRunHook {
```

```
 var process: Option[Process] = None
```

```
 override def beforeStarted(): Unit = {
```

```
 Process("grunt dist", base).run
```

```
 }
```

```
 override def afterStarted(addr: InetSocketAddress): Unit = {
```

```
 process = Some(Process("grunt watch", base).run)
```

```
 }
```

```
 override def afterStopped(): Unit = {
```

```
 process.map(p => p.destroy())
```

```
 process = None
```

```
 }
```

```
 }
```

```
 GruntProcess
```

```
 }
```

```
}
```

Once the application has been started we execute `grunt watch` and when the application has been stopped we destroy the grunt process. There's nothing to change in `build.sbt`

## Add compiler options

For example, you may want to add the feature flag to have details on feature warnings:

```
[info] Compiling 1 Scala source to ~/target/scala-2.10/classes...
```

```
[warn] there were 1 feature warnings; re-run with -feature for details
```

```
Simply add -feature to the scalacOptions attribute:
```

```
scalacOptions += "-feature"
```

## Add additional asset directory

For example you can add the `pictures` folder to be included as an additional asset directory:

```
unmanagedResourceDirectories in Assets <+= baseDirectory { _ / "pictures" }
```

This will allow you to use `routes.Assets.at` with this folder.

## Disable documentation

To speed up compilation you can disable documentation generation:

```
sources in (Compile, doc) := Seq.empty
```

```
publishArtifact in (Compile, packageDoc) := false
```

The first line will disable documentation generation and the second one will avoid to publish the documentation artifact.

## Configure ivy logging level

By default `ivyLoggingLevel` is set on `UpdateLogging.DownloadOnly`. You can change this value with:

- `UpdateLogging.Quiet` only displays errors
- `UpdateLogging.FULL` logs the most

For example if you want to only display errors:

```
ivyLoggingLevel := UpdateLogging.Quiet
```

## Fork and parallel execution in test

By default parallel execution is disabled and fork is enabled. You can change this behavior by setting `parallelExecution` in `Test` and/or `fork` in `Test`:

```
parallelExecution in Test := true
```

```
fork in Test := false
```

**Next:** Debugging your build

## Debugging your build

If you are having difficulties getting sbt to do what you want it to do, you may need to use some of the built in utilities that sbt provides to help you debug your build.

# Debugging dependencies

By default, sbt generates reports of all your dependencies, including dependency trees to show which dependencies transitively brought in other dependencies, and conflict resolution tables showing how sbt decided which version of a dependency it selected when multiple were requested.

The reports are generated into xml files, with an accompanying XSL stylesheet that allow browsers that support XSL to convert the XML reports into HTML. Browsers with this support include Firefox and Safari, and notably don't include Chrome.

The reports can be found in the `target/resolution-cache/reports` directory of your project, one is generated for each scope in your project, and are named `organization-projectId.scalaVersion-scope.xml`, for example, `com.example-my-first-app_2.11-compile.xml`. When opened in Firefox, this report looks something like this:

**my-first-app\_2.11 1.0-SNAPSHOT by com.example**  
resolved on 2015-04-07 13:03:12



[compile](#) [runtime](#) [test](#) [provided](#) [optional](#) [compile-internal](#) [runtime-internal](#) [test-internal](#) [plugin](#) [sources](#) [docs](#) [pom](#) [scala-tool](#)

**Dependencies Stats**

Modules	51
Revisions	60 (0 searched, 0 downloaded, 9 evicted, 0 errors)
Artifacts	51 (0 downloaded, 0 failed)
Artifacts size	33952 kB (0 kB downloaded, 33952 kB in cache)

**Conflicts**

Module	Selected	Evicted
guava by com.google.guava	16.0.1	15.0
commons-codec by commons-codec	1.9	1.3
slf4j-api by org.slf4j	1.7.6	1.6.1 1.7.2 1.7.5
netty by io.netty	3.9.3.Final	3.9.2.Final 3.6.3.Final
jackson-annotations by com.fasterxml.jackson.core	2.3.2	2.3.0
scala-library by org.scala-lang	2.11.1	2.11.0

**Dependencies Overview**

Module	Revision	Status	Resolver	Default	Licenses	Size
play-ws_2.11 by com.typesafe.play	2.3.8	release	sbt-chain	false	Apache-2.0	586 kB
.... signpost-commonhttp4 by math.signpost	1.2.1.2	release	sbt-chain	false		6 kB
.... httpclient by org.apache.httpcomponents	4.0.1	release	sbt-chain	false	Apache License	284 kB
.... httpcore by org.apache.httpcomponents	4.0.1	release	sbt-chain	false	Apache License	169 kB
.... commons-codec by commons-codec	1.9	release	sbt-chain	false		258 kB
.... commons-codec by commons-codec	1.3					0 kB
.... commons-logging by commons-logging	1.1.1	release	sbt-chain	false		59 kB
.... httpcore by org.apache.httpcomponents	4.0.1	release	sbt-chain	false	Apache License	169 kB

# Debugging settings

There are a few useful commands that sbt provides that can be used to understand your build and work out where things may be going wrong.

## The show command

The show command shows the return value from any sbt task. So for example, if you're not sure if a certain source file is being compiled or not, you can run `show sources` to see if sbt is including it in the sources:

```
[my-first-app] $ show sources
[info] ArrayBuffer(my-first-app/app/controllers/Application.scala,
 my-first-app/target/scala-2.11/twirl/main/views/html/index.template.scala,
 my-first-app/target/scala-2.11/twirl/main/views/html/main.template.scala,
 my-first-app/target/scala-2.11/src_managed/main/routes_reverseRouting.scala,
 my-first-app/target/scala-2.11/src_managed/main/routes_routing.scala,
 my-first-app/target/scala-2.11/src_managed/main/controllers/routes.java)
```

The output above has been formatted to ensure it fits cleanly on the screen, you may need to copy it to an editor to make sense of it if the task you run returns a long list of items.

You can also specify a task a particular scope, eg `test:sources` or `compile:sources`, or for a particular project, `my-project/compile:sources`, and in some cases, where tasks are scoped by another task, you can specify that scope too, for example, to see everything that will be packaged into your projects jar file, you want to show the `mappings` task, scoped to the `packageBin` task:

```
[my-first-app] $ show compile:packageBin::mappings
[info] List(
 (my-first-app/target/scala-2.11/classes/application.conf,application.conf),
 (my-first-app/target/scala-2.11/classes/controllers/Application.class,controllers/Application.class),
 ...)
```

## The inspect command

The inspect command gives you detailed information about a task, including what it depends on, what depends on it, where it was defined, etc. It can be used like the `show` command:

```
[my-first-app] $ inspect managedSources
[info] Task: scala.collection.Seq[java.io.File]
[info] Description:
[info] Sources generated by the build.
[info] Provided by:
[info] {file:my-first-app/}root/compile:managedSources
[info] Defined at:
[info] (sbt.Defaults) Defaults.scala:185
[info] Dependencies:
[info] compile:sourceGenerators
[info] Reverse dependencies:
[info] compile:sources
...)
```

Here we've inspected the `managedSources` command, it tells us that this is a task that produces a sequence of files, it has a description of `Sources generated by the`

`build`. You can see that it depends on the `sourceGenerators` task, and the `sources` task depends on it. You can also see where it was defined, in this case, it's from sbt's default task definitions, line 185.

## The inspect tree command

The inspect tree command shows a whole tree of task dependencies for a particular task. If we inspect the tree for the `unmanagedSources` task, we can see it here:

```
[my-first-app] $ inspect tree unmanagedSources
[info] compile:unmanagedSources = Task[scala.collection.Seq[java.io.File]]
[info] +-/*:sourcesInBase = true
[info] +-/*:unmanagedSources::includeFilter = sbt.SimpleFilter@3dc46f24
[info] +-compile:unmanagedSourceDirectories = List(my-first-app/app, my-first-a...
[info] | +-compile:javaSource = app
[info] || +-/*:baseDirectory = my-first-app
[info] || | +-/*:thisProject = Project(id root, base: my-first-app, configurations: List(compile, ...
[info] ||
[info] | +-compile:scalaSource = app
[info] | | +-/*:baseDirectory = my-first-app
[info] | | | +-/*:thisProject = Project(id root, base: my-first-app, configurations: List(compile, ...
[info] |
[info] +-/*:baseDirectory = my-first-app
[info] +-/*:excludeFilter = sbt.HiddenFileFilter$@49e479da
```

This shows the whole tree of tasks that sbt uses to discover the sources in your project, including the filters to decide which files to be included or excluded. The `inspect tree` command is particularly useful when you're not sure how some part of your build is structured, and you want to find out how it all fits together so you can then dive in deeper.

# Debugging incremental compilation

A common problem that people have in Play is they find Play recompiles and reloads when they don't expect it to. This is often caused by source generators or IDEs that inadvertently update elements of Play's classpath, forcing a reload. To debug problems like this, we can look at the debug log from the compile task. When sbt runs a task it captures all the log output, whether it displays it or not, so that you can inspect it later if you want. It can be inspected using the `last` command.

So, let's say you `compile`, and a file needs to be recompiled:

```
[my-first-app] $ compile
[info] Compiling 1 Scala source to my-first-app/target/scala-2.11/classes...
[success] Total time: 1 s, completed 07/04/2015 1:28:43 PM
```

You can get a full debug log of what happened during the `compile` command by running `last compile`. This will dump a lot of output, but only the first part is what we are interested in, shown here:

```
[my-first-app] $ last compile
[debug]
```

```
[debug] Initial source changes:
[debug] removed: Set()
[debug] added: Set()
[debug] modified: Set(my-first-app/app/controllers/Application.scala)
[debug] Removed products: Set()
[debug] External API changes: API Changes: Set()
[debug] Modified binary dependencies: Set()
[debug] Initial directly invalidated sources: Set(my-first-app/app/controllers/Application.scala)
[debug]
[debug] Sources indirectly invalidated by:
[debug] product: Set()
[debug] binary dep: Set()
[debug] external source: Set()
```

What this tells us is that a recompile was triggered because `my-first-app/app/controllers/Application.scala` was modified.

Next: [Working with public assets](#)

# Working with public assets

This section covers serving your application's static resources such as JavaScript, CSS and images.

Serving a public resource in Play is the same as serving any other HTTP request. It uses the same routing as regular resources using the controller/action path to distribute CSS, JavaScript or image files to the client.

## The public/ folder

By convention public assets are stored in the `public` folder of your application. This folder can be organized the way that you prefer. We recommend the following organization:

```
public
└─ javascripts
└─ stylesheets
└─ images
```

If you follow this structure it will be simpler to get started, but nothing stops you to modifying it once you understand how it works.

## WebJars

[WebJars](#) provide a convenient and conventional packaging mechanism that is a part of Activator and sbt. For example you can declare that you will be using the popular[Bootstrap library](#) simply by adding the following dependency in your build file:

```
libraryDependencies += "org.webjars" % "bootstrap" % "3.3.4"
```

WebJars are automatically extracted into a `lib` folder relative to your public assets for convenience. For example, if you declared a dependency on [RequireJs](#) then you can reference it from a view using a line like:

```
<script data-main="@routes.Assets.at("javascripts/main.js")" type="text/javascript"
src="@routes.Assets.at("lib/requirejs/require.js")"></script>
```

Note the `lib/requirejs/require.js` path. The `lib` folder denotes the extract WebJar assets, the `requirejs` folder corresponds to the WebJar artifactId, and the `require.js` refers to the required asset at the root of the WebJar.

## How are public assets packaged?

During the build process, the contents of the `public` folder are processed and added to the application classpath.

When you package your application, all assets for the application, including all sub projects, are aggregated into a single jar, in `target/my-first-app-1.0.0-assets.jar`. This jar is included in the distribution so that your Play application can serve them. This jar can also be used to deploy the assets to a CDN or reverse proxy.

## The Assets controller

Play comes with a built-in controller to serve public assets. By default, this controller provides caching, ETag, gzip and compression support.

The controller is available in the default Play JAR as `controllers.Assets` and defines a single `at` action with two parameters:

```
Assets.at(path: String, file: String)
```

The `path` parameter must be fixed and defines the directory managed by the action.

The `file` parameter is usually dynamically extracted from the request path.

Here is the typical mapping of the `Assets` controller in your `conf/routes` file:

```
GET /assets/*file controllers.Assets.at(path="/public", file)
```

Note that we define the `*file` dynamic part that will match the `.*` regular expression. So for example, if you send this request to the server:

```
GET /assets/javascripts/jquery.js
```

The router will invoke the `Assets.at` action with the following parameters:

```
controllers.Assets.at("/public", "javascripts/jquery.js")
```

This action will look-up and serve the file and if it exists.

## Reverse routing for public assets

As for any controller mapped in the routes file, a reverse controller is created in `controllers.routes.Assets`. You use this to reverse the URL needed to fetch a public resource. For example, from a template:

```
<script src="@routes.Assets.at("javascripts/jquery.js")"></script>
```

This will produce the following result:

```
<script src="/assets/javascripts/jquery.js"></script>
```

Note that we don't specify the first `folder` parameter when we reverse the route. This is because our routes file defines a single mapping for the `Assets.at` action, where the `folder` parameter is fixed. So it doesn't need to be specified.

However, if you define two mappings for the `Assets.at` action, like this:

```
GET /javascripts/*file controllers.Assets.at(path="/public/javascripts", file)
GET /images/*file controllers.Assets.at(path="/public/images", file)
```

You will then need to specify both parameters when using the reverse router:

```
<script src="@routes.Assets.at("/public/javascripts", "jquery.js")"></script>

```

## Reverse routing and fingerprinting for public assets

[sbt-web](#) brings the notion of a highly configurable asset pipeline to Play e.g. in your build file: `pipelineStages := Seq(rjs, digest, gzip)`

The above will order the RequireJs optimizer (`sbt-rjs`), the digester (`sbt-digest`) and then compression (`sbt-gzip`). Unlike many sbt tasks, these tasks will execute in the order declared, one after the other.

In essence asset fingerprinting permits your static assets to be served with aggressive caching instructions to a browser. This will result in an improved experience for your users given that subsequent visits to your site will result in less assets requiring to be downloaded. Rails also describes the benefits of [asset fingerprinting](#).

The above declaration of `pipelineStages` and the requisite `addSbtPlugin` declarations in your `plugins.sbt` for the plugins you require are your start point. You must then declare to Play what assets are to be versioned. The following routes file entry declares that all assets are to be versioned:

```
GET /assets/*file controllers.Assets.versioned(path="/public", file: Asset)
```

Make sure you indicate that `file` is an asset by writing `file: Asset`.

You then use the reverse router, for example within a `scala.html` view:

```
<link rel="stylesheet" href="@routes.Assets.versioned("assets/css/app.css")">
```

We highly encourage the use of asset fingerprinting.

---

## Etag support

The `Assets` controller automatically manages **ETag** HTTP Headers. The ETag value is generated from the digest (if `sbt-digest` is being used in the asset pipeline) or otherwise the resource name and the file's last modification date. If the resource file is embedded into a file, the JAR file's last modification date is used.

When a web browser makes a request specifying this **Etag** then the server can respond with **304 NotModified**.

---

## Gzip support

If a resource with the same name but using a `.gz` suffix is found then the `Assets` controller will also serve the latter and add the following HTTP header:

`Content-Encoding: gzip`

Including the `sbt-gzip` plugin in your build and declaring its position in the `pipelineStages` is all that is required to generate gzip files.

---

## Additional Cache-Control directive

Using Etag is usually enough for the purposes of caching. However if you want to specify a custom `Cache-Control` header for a particular resource, you can specify it in your `application.conf` file. For example:

```
Assets configuration
~~~~~
"assets.cache./public/stylesheets/bootstrap.min.css"="max-age=3600"
```

## Managed assets

Starting with Play 2.3 managed assets are processed by [sbt-web](#) based plugins. Prior to 2.3 Play bundled managed asset processing in the form of CoffeeScript, LESS, JavaScript linting (ClosureCompiler) and RequireJS optimization. The following sections describe sbt-web and how the equivalent 2.2 functionality can be achieved. Note though that Play is not limited to this asset processing technology as many plugins should become available to sbt-

web over time. Please check-in with the [sbt-web](#) project to learn more about what plugins are available.

Many plugins use sbt-web's [js-engine plugin](#). js-engine is able to execute plugins written to the Node API either within the JVM via the excellent [Trireme](#) project, or directly on [Node.js](#) for superior performance. Note that these tools are used during the development cycle only and have no involvement during the runtime execution of your Play application. If you have Node.js installed then you are encouraged to declare the following environment variable. For Unix, if `SBT_OPTS` has been defined elsewhere then you can:

```
export SBT_OPTS="$SBT_OPTS -Dsbt.jse.engineType=Node"
```

The above declaration ensures that Node.js is used when executing any sbt-web plugin.

Next: [Using CoffeeScript](#)

# Using CoffeeScript

[CoffeeScript](#) is a small and elegant language that compiles into JavaScript. It provides a nice syntax for writing JavaScript code.

Compiled assets in Play must be defined in the `app/assets` directory. They are handled by the build process and CoffeeScript sources are compiled into standard JavaScript files. The generated JavaScript files are distributed as standard resources into the same `public/` folder as other unmanaged assets, meaning that there is no difference in the way you use them once compiled.

For example a CoffeeScript source file `app/assets/javascripts/main.coffee` will be available as a standard JavaScript resource, at `public/javascripts/main.js`. CoffeeScript sources are compiled automatically during an `assets` command, or when you refresh any page in your browser while you are running in development mode. Any compilation errors will be displayed in your browser:

A screenshot of a web browser window titled "Compilation error". The address bar shows "localhost:9000". The main content area has a red header with the text "Compilation error". Below the header, the error message "unclosed ( on line 9" is displayed. A code editor shows the following CoffeeScript code:

```
5 # Conditions:
6 number = -42 if opposite
7
8 # Functions:
9 square = ((x) -> x * x
10
11 # Arrays:
12 list = [1, 2, 3, 4, 5]
13
```

# Layout

Here is an example layout for using CoffeeScript in your projects:

```
app
└ assets
 └ javascripts
 └ main.coffee
```

You can use the following syntax to use the compiled JavaScript file in your template:

```
<script src="@routes.Assets.at("javascripts/main.js")">
```

---

# Enablement and Configuration

CoffeeScript compilation is enabled by simply adding the plugin to your `plugins.sbt` file when using the `PlayJava` or `PlayScala` plugins:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-coffeescript" % "1.0.0")
```

The plugin's default configuration is normally sufficient. However please refer to the [plugin's documentation](#) for information on how it may be configured.

**Next: Using LESS CSS**

# Using LESS CSS

**LESS CSS** is a dynamic stylesheet language. It allows considerable flexibility in the way you write CSS files including support for variables, mixins and more.

Compilable assets in Play must be defined in the `app/assets` directory. They are handled by the build process, and LESS sources are compiled into standard CSS files. The generated CSS files are distributed as standard resources into the same `public/` folder as the unmanaged assets, meaning that there is no difference in the way you use them once compiled.

For example, a LESS source file at `app/assets/stylesheets/main.less` will be available as a standard resource at `public/stylesheets/main.css`. Play will compile `main.less` automatically. Other LESS files need to be included in your `build.sbt` file:

```
includeFilter in (Assets, LessKeys.less) := "foo.less" | "bar.less"
```

LESS sources are compiled automatically during an `assets` command, or when you refresh any page in your browser while you are running in development mode. Any compilation errors will be displayed in your browser:

A screenshot of a web browser window titled "Compilation error". The address bar shows "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below this, the error message "variable @main-colo is undefined" is displayed. A stack trace follows, indicating the error occurred in file "/Volumes/Data/gbo/myFirstApp/app/assets/stylesheets/main.less" at line 4. The code shown is:

```
1 @main-color: red;
2
3 h1 {
4 color: @main-colo;
5 }
```

## Working with partial LESS source files

You can split your LESS source into several libraries and use the LESS `import` feature. To prevent library files from being compiled individually (or imported) we need them to be skipped by the compiler. To do this partial source files can be prefixed with the underscore (`_`) character, for example: `_myLibrary.less`. The following configuration enables the compiler to ignore partials:

```
includeFilter in (Assets, LessKeys.less) := "*.less"
```

```
excludeFilter in (Assets, LessKeys.less) := "_*.less"
```

# Layout

Here is an example layout for using LESS in your project:

```
app
└ assets
 └ stylesheets
 └ main.less
 └ utils
 └ reset.less
 └ layout.less
```

With the following `main.less` source:

```
@import "utils/reset.less";
@import "utils/layout.less";
```

```
h1 {
 color: red;
}
```

The resulting CSS file will be compiled as `public/stylesheets/main.css` and you can use this in your template as any regular public asset.

```
<link rel="stylesheet" href="@routes.Assets.at("stylesheets/main.css")">
```

## Using LESS with Bootstrap

Bootstrap is a very popular library used in conjunction with LESS.

To use Bootstrap you can use its WebJar by adding it to your library dependencies. For example, within a `build.sbt` file:

```
libraryDependencies += "org.webjars" % "bootstrap" % "3.3.4"
```

sbt-web will automatically extract WebJars into a lib folder relative to your asset's target folder. Therefore to use Bootstrap you can import relatively e.g.:

```
@import "lib/bootstrap/less/bootstrap.less";

h1 {
 color: @font-size-h1;
}
```

## Enablement and Configuration

LESS compilation is enabled by simply adding the plugin to your `plugins.sbt` file when using the `PlayJava` or `PlayScala` plugins:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-less" % "1.0.6")
```

The plugin's default configuration is normally sufficient. However please refer to the [plugin's documentation](#) for information on how it may be configured.

[Next: Using JSHint](#)

# Using JSHint

From its [website documentation](#):

JSHint is a community-driven tool to detect errors and potential problems in JavaScript code and to enforce your team's coding conventions. It is very flexible so you can easily adjust it to your particular coding guidelines and the environment you expect your code to execute in.

Any JavaScript file present in `app/assets` will be processed by JSHint and checked for errors.

## Check JavaScript sanity

JavaScript code is compiled during the `assets` command as well as when the browser is refreshed during development mode. Errors are shown in the browser just like any other compilation error.

## Enablement and Configuration

JSHint processing is enabled by simply adding the plugin to your `plugins.sbt` file when using the `PlayJava` or `PlayScala` plugins:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-jshint" % "1.0.3")
```

The plugin's default configuration is normally sufficient. However please refer to the [plugin's documentation](#) for information on how it may be configured.

[Next: Using RequireJS](#)

# RequireJS

According to [RequireJS](#)' website

RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments... Using a modular script loader like RequireJS will improve the speed and quality of your code.

What this means in practice is that one can use [RequireJS](#) to modularize your JavaScript. RequireJS achieves this by implementing a semi-standard API called [Asynchronous Module](#)

Definition (other similar ideas include [CommonJS](#)). Using AMD makes it is possible to resolve and load javascript modules on the *client side* while allowing server side *optimization*. For server side optimization module dependencies may be minified and combined using [UglifyJS 2](#).

By convention RequireJS expects a main.js file to bootstrap its module loader.

## Deployment

The RequireJS optimizer shouldn't generally kick-in until it is time to perform a deployment i.e. by running the `start`, `stage` or `dist` tasks.

If you're using WebJars with your build then the RequireJS optimizer plugin will also ensure that any JavaScript resources referenced from within a WebJar are automatically referenced from the [jsdelivr](#) CDN. In addition if any `.min.js` file is found then that will be used in place of `.js`. An added bonus here is that there is no change required to your html!

## Enablement and Configuration

RequireJS optimization is enabled by simply adding the plugin to your `plugins.sbt` file when using the `PlayJava` or `PlayScala` plugins:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-rjs" % "1.0.7")
```

To add the plugin to the asset pipeline you can declare it as follows (assuming just the one plugin for the pipeline - add others into the sequence such as digest and gzip as required):

```
pipelineStages := Seq(rjs)
```

A standard build profile for the RequireJS optimizer is provided and should suffice for most projects. However please refer to the [plugin's documentation](#) for information on how it may be configured.

Note that RequireJS performs a lot of work and while it works when executed in-JVM under Trirreme, you will be best to use Node.js as the js-engine from a performance perspective. For convenience you can set the `sbt.jse.engineType` property in `SBT_OPTS`. For example on Unix:

```
export SBT_OPTS="$SBT_OPTS -Dsbt.jse.engineType=Node"
```

Please refer to the [plugin's documentation](#) for information on how it may be configured.

Next: [Configuration](#)

# Configuration file syntax and features

The configuration file used by Play is based on the [Typesafe config library](#).

The configuration file of a Play application must be defined in `conf/application.conf`.

It uses the [HOCON format](#).

As well as the `application.conf` file, configuration comes from a couple of other places.

- Default settings are loaded from any `reference.conf` files found on the classpath. Most Play JARs include a `reference.conf` file with default settings. Settings in `application.conf` will override settings in `reference.conf` files.
- It's also possible to set configuration using system properties. System properties override `application.conf` settings.

## Specifying an alternative configuration file

At runtime, the default `application.conf` is loaded from the classpath. System properties can be used to force a different config source:

- `config.resource` specifies a resource name including the extension, i.e. `application.conf` and not just `application`
  - `config.file` specifies a filesystem path, again it should include the extension, not be a basename
- These system properties specify a replacement for `application.conf`, not an addition. If you still want to use some values from the `application.conf` file then you can include the `application.conf` in your other `.conf` file by writing `include "application"` at the top of that file. After you've included the `application.conf`'s settings in your new `.conf` file you can then specify any settings that you want override.

## Using with Akka

Akka will use the same configuration file as the one defined for your Play application. Meaning that you can configure anything in Akka in the `application.conf` directory. In Play, Akka reads its settings from within the `play.akka` setting, not from the `akka` setting.

## Using with the `run` command

There are a couple of special things to know about configuration when running your application with the `run` command.

## Extra `devSettings`

You can configure extra settings for the `run` command in your `build.sbt`. These settings won't be used when you deploy your application.

```
devSettings := Map("play.server.http.port" -> "8080")
```

## HTTP server settings in `application.conf`

In `run` mode the HTTP server part of Play starts before the application has been compiled. This means that the HTTP server cannot access the `application.conf` file when it starts. If you want to override HTTP server settings while using the `run` command you cannot use the `application.conf` file. Instead, you need to either use system properties or the `devSettings` setting shown above. An example of a server setting is the HTTP port. Other server settings can be seen [here](#).

```
> run -Dhttp.port=1234
```

# HOCON Syntax

HOCON has similarities to JSON; you can find the JSON spec at <http://json.org/> of course.

## Unchanged from JSON

- files must be valid UTF-8
- quoted strings are in the same format as JSON strings
- values have possible types: string, number, object, array, boolean, null
- allowed number formats matches JSON; as in JSON, some possible floating-point values are not represented, such as `NaN`

## Comments

Anything between `//` or `#` and the next newline is considered a comment and ignored, unless the `//` or `#` is inside a quoted string.

## Omit root braces

JSON documents must have an array or object at the root. Empty files are invalid documents, as are files containing only a non-array non-object value such as a string.

In HOCON, if the file does not begin with a square bracket or curly brace, it is parsed as if it were enclosed with `{}` curly braces.

A HOCON file is invalid if it omits the opening `{` but still has a closing `}`; the curly braces must be balanced.

## Key-value separator

The `=` character can be used anywhere JSON allows `:`, i.e. to separate keys from values. If a key is followed by `{`, the `:` or `=` may be omitted. So `"foo" { }` means `"foo" : {}"`

## Commas

Values in arrays, and fields in objects, need not have a comma between them as long as they have at least one ASCII newline (`\n`, decimal value 10) between them.

The last element in an array or last field in an object may be followed by a single comma. This extra comma is ignored.

- `[1, 2, 3, ]` and `[1, 2, 3]` are the same array.
- `[1\n2\n3]` and `[1, 2, 3]` are the same array.
- `[1, 2, 3,, ]` is invalid because it has two trailing commas.
- `[, 1, 2, 3]` is invalid because it has an initial comma.
- `[1,, 2, 3]` is invalid because it has two commas in a row.
- these same comma rules apply to fields in objects.

## Duplicate keys

The JSON spec does not clarify how duplicate keys in the same object should be handled. In HOCON, duplicate keys that appear later override those that appear earlier, unless both values are objects. If both values are objects, then the objects are merged.

Note: this would make HOCON a non-superset of JSON if you assume that JSON requires duplicate keys to have a behavior. The assumption here is that duplicate keys are invalid JSON.

To merge objects:

- add fields present in only one of the two objects to the merged object.
- for non-object-valued fields present in both objects, the field found in the second object must be used.
- for object-valued fields present in both objects, the object values should be recursively merged according to these same rules.

Object merge can be prevented by setting the key to another value first. This is because merging is always done two values at a time; if you set a key to an object, a non-object, then an object, first the non-object falls back to the object (non-object always wins), and then the object falls back to the non-object (no merging, object is the new value). So the two objects never see each other.

These two are equivalent:

```
{
 "foo" : { "a" : 42 },
 "foo" : { "b" : 43 }
}

{
 "foo" : { "a" : 42, "b" : 43 }
}
```

And these two are equivalent:

```
{
 "foo" : { "a" : 42 },
 "foo" : null,
 "foo" : { "b" : 43 }
}

{
 "foo" : { "b" : 43 }
}
```

The intermediate setting of `"foo"` to `null` prevents the object merge.

## Paths as keys

If a key is a path expression with multiple elements, it is expanded to create an object for each path element other than the last. The last path element, combined with the value, becomes a field in the most-nested object.

In other words:

`foo.bar : 42`

is equivalent to:

`foo { bar : 42 }`

and:

`foo.bar.baz : 42`

is equivalent to:

`foo { bar { baz : 42 } }`

and so on. These values are merged in the usual way; which implies that:

`a.x : 42, a.y : 43`

is equivalent to:

`a { x : 42, y : 43 }`

Because path expressions work like value concatenations, you can have whitespace in keys:

`a b c : 42`

is equivalent to:

`"a b c" : 42`

Because path expressions are always converted to strings, even single values that would normally have another type become strings.

- `true : 42` is `"true" : 42`
- `3.14 : 42` is `"3.14" : 42`

As a special rule, the unquoted string `include` may not begin a path expression in a key, because it has a special interpretation (see below).

---

# Substitutions

Substitutions are a way of referring to other parts of the configuration tree.

The syntax is  `${pathexpression}` or  `${?pathexpression}` where the `pathexpression` is a path expression as described above. This path expression has the same syntax that you could use for an object key.

The `?` in  `${?pathexpression}` must not have whitespace before it; the three characters  `${?}` must be exactly like that, grouped together.

For substitutions which are not found in the configuration tree, implementations may try to resolve them by looking at system environment variables or other external sources of configuration. (More detail on environment variables in a later section.)

Substitutions are not parsed inside quoted strings. To get a string containing a substitution, you must use value concatenation with the substitution in the unquoted portion:

```
key : ${animal.favorite} is my favorite animal
```

Or you could quote the non-substitution portion:

```
key : ${animal.favorite}" is my favorite animal"
```

Substitutions are resolved by looking up the path in the configuration. The path begins with the root configuration object, i.e. it is “absolute” rather than “relative.”

Substitution processing is performed as the last parsing step, so a substitution can look forward in the configuration. If a configuration consists of multiple files, it may even end up retrieving a value from another file. If a key has been specified more than once, the substitution will always evaluate to its latest-assigned value (the merged object or the last non-object value that was set).

If a configuration sets a value to `null` then it should not be looked up in the external source. Unfortunately there is no way to “undo” this in a later configuration file; if you have `{ "HOME" : null }` in a root object, then  `${HOME}` will never look at the

environment variable. There is no equivalent to JavaScript's `delete` operation in other words.

If a substitution does not match any value present in the configuration and is not resolved by an external source, then it is undefined. An undefined substitution with the  `${foo}` syntax is invalid and should generate an error.

If a substitution with the  `${?foo}` syntax is undefined:

- if it is the value of an object field then the field should not be created. If the field would have overridden a previously-set value for the same field, then the previous value remains.
- if it is an array element then the element should not be added.
- if it is part of a value concatenation then it should become an empty string.
- `foo : ${?bar}` would avoid creating field `foo` if `bar` is undefined, but `foo : ${?bar} ${?baz}` would be a value concatenation so if `bar` or `baz` are not defined, the result is an empty string.

Substitutions are only allowed in object field values and array elements (value concatenations), they are not allowed in keys or nested inside other substitutions (path expressions).

A substitution is replaced with any value type (number, object, string, array, true, false, null). If the substitution is the only part of a value, then the type is preserved. Otherwise, it is value-concatenated to form a string.

Circular substitutions are invalid and should generate an error.

Implementations must take care, however, to allow objects to refer to paths within themselves. For example, this must work:

```
bar : { foo : 42,
 baz : ${bar.foo}
 }
```

Here, if an implementation resolved all substitutions in `bar` as part of resolving the substitution  `${bar.foo}`, there would be a cycle. The implementation must only resolve the `foo` field in `bar`, rather than recursing the entire `bar` object.

# Includes

## Include syntax

An *include statement* consists of the unquoted string `include` and a single quoted string immediately following it. An include statement can appear in place of an object field. If the unquoted string `include` appears at the start of a path expression where an object key would be expected, then it is not interpreted as a path expression or a key.

Instead, the next value must be a *quoted* string. The quoted string is interpreted as a filename or resource name to be included.

Together, the unquoted `include` and the quoted string substitute for an object field syntactically, and are separated from the following object fields or includes by the usual comma (and as usual the comma may be omitted if there's a newline).

If an unquoted `include` at the start of a key is followed by anything other than a single quoted string, it is invalid and an error should be generated.

There can be any amount of whitespace, including newlines, between the unquoted `include` and the quoted string.

Value concatenation is NOT performed on the “argument” to `include`. The argument must be a single quoted string. No substitutions are allowed, and the argument may not be an unquoted string or any other kind of value.

Unquoted `include` has no special meaning if it is not the start of a key’s path expression.

It may appear later in the key:

```
this is valid
{ foo include : 42 }
equivalent to
{ "foo include" : 42 }
```

It may appear as an object or array value:

```
{ foo : include } # value is the string "include"
[include] # array of one string "include"
```

You can quote `"include"` if you want a key that starts with the word `"include"`, only unquoted `include` is special:

```
{ "include" : 42 }
```

## Include semantics: merging

An *including file* contains the `include` statement and an *included file* is the one specified in the `include` statement. (They need not be regular files on a filesystem, but assume they are for the moment.)

An included file must contain an object, not an array. This is significant because both JSON and HOCON allow arrays as root values in a document.

If an included file contains an array as the root value, it is invalid and an error should be generated.

The included file should be parsed, producing a root object. The keys from the root object are conceptually substituted for the `include` statement in the including file.

- If a key in the included object occurred prior to the `include` statement in the including object, the included key’s value

- overrides or merges with the earlier value, exactly as with duplicate keys found in a single file.
- If the including file repeats a key from an earlier-included object, the including file's value would override or merge with the one from the included file.

## Include semantics: substitution

Substitutions in included files are looked up at two different paths; first, relative to the root of the included file; second, relative to the root of the including configuration.

Recall that substitution happens as a final step, *after* parsing. It should be done for the entire app's configuration, not for single files in isolation.

Therefore, if an included file contains substitutions, they must be “fixed up” to be relative to the app's configuration root.

Say for example that the root configuration is this:

```
{ a : { include "foo.conf" } }
```

And “foo.conf” might look like this:

```
{ x : 10, y : ${x} }
```

If you parsed “foo.conf” in isolation, then  `${x}` would evaluate to 10, the value at the path `x`. If you include “foo.conf” in an object at key `a`, however, then it must be fixed up to be  `${a.x}` rather than  `${x}`.

Say that the root configuration redefines `a.x`, like this:

```
{
 a : { include "foo.conf" }
 a : { x : 42 }
}
```

Then the  `${x}` in “foo.conf”, which has been fixed up to  `${a.x}`, would evaluate to 42 rather than to 10. Substitution happens *after* parsing the whole configuration.

However, there are plenty of cases where the included file might intend to refer to the application's root config. For example, to get a value from a system property or from the reference configuration. So it's not enough to only look up the “fixed up” path, it's necessary to look up the original path as well.

## Include semantics: missing files

If an included file does not exist, the include statement should be silently ignored (as if the included file contained only an empty object).

## Include semantics: locating resources

Conceptually speaking, the quoted string in an include statement identifies a file or other resource “adjacent to” the one being parsed and of the same type as the one being parsed. The meaning of “adjacent to”, and the string itself, has to be specified separately for each kind of resource.

Implementations may vary in the kinds of resources they support including.

On the Java Virtual Machine, if an include statement does not identify anything “adjacent to” the including resource, implementations may wish to fall back to a classpath resource. This allows configurations found in files or URLs to access classpath resources.

For resources located on the Java classpath:

- included resources are looked up by calling `getResource()` on the same class loader used to look up the including resource.
- if the included resource name is absolute (starts with '/') then it should be passed to `getResource()` with the '/' removed.
- if the included resource name does not start with '/' then it should have the “directory” of the including resource prepended to it, before passing it to `getResource()`. If the including resource is not absolute (no '/') and has no “parent directory” (is just a single path element), then the included relative resource name should be left as-is.
- it would be wrong to use `getResource()` to get a URL and then locate the included name relative to that URL, because a class loader is not required to have a one-to-one mapping between paths in its URLs and the paths it handles in `getResource()`. In other words, the “adjacent to” computation should be done on the resource name not on the resource’s URL.

For plain files on the filesystem:

- if the included file is an absolute path then it should be kept absolute and loaded as such.
- if the included file is a relative path, then it should be located relative to the directory containing the including file. The current working directory of the process parsing a file must NOT be used when interpreting included paths.
- if the file is not found, fall back to the classpath resource. The classpath resource should not have any package name added in front, it should be relative to the “root”; which means any leading ‘/’ should just be removed (absolute is the same as relative since it’s root-relative). The ‘/’ is handled for

consistency with including resources from inside other classpath resources, where the resource name may not be root-relative and “/” allows specifying relative to root.

URLs:

- for both filesystem files and Java resources, if the included name is a URL (begins with a protocol), it would be reasonable behavior to try to load the URL rather than treating the name as a filename or resource name.
  - for files loaded from a URL, “adjacent to” should be based on parsing the URL’s path component, replacing the last path element with the included name.
  - file: URLs should behave in exactly the same way as a plain filename
- 

## Duration format

The supported unit strings for duration are case sensitive and must be lowercase. Exactly these strings are supported:

- ns, nanosecond, nanoseconds
  - us, microsecond, microseconds
  - ms, millisecond, milliseconds
  - s, second, seconds
  - m, minute, minutes
  - h, hour, hours
  - d, day, days
- 

## Size in bytes format

For single bytes, exactly these strings are supported:

- B, b, byte, bytes

For powers of ten, exactly these strings are supported:

- kB, kilobyte, kilobytes
- MB, megabyte, megabytes
- GB, gigabyte, gigabytes
- TB, terabyte, terabytes
- PB, petabyte, petabytes
- EB, exabyte, exabytes

- ZB, zettabyte, zettabytes
- YB, yottabyte, yottabytes

For powers of two, exactly these strings are supported:

- K, k, KiB, kibibyte, kibibytes
- M, m, MiB, mebibyte, mebibytes
- G, g, GiB, gibibyte, gibibytes
- T, t, TiB, tebibyte, tebibytes
- P, p, PiB, pebibyte, pebibytes
- E, e, EiB, exbibyte, exbibytes
- Z, z, ZiB, zebibyte, zebibytes
- Y, y, YiB, yobibyte, yobibytes

## Conventional override by system properties

Java system properties override settings found in the `application.conf` and `reference.conf` files. This supports specifying config options on the command line. ie.`play -Dkey=value run`

Note : Play forks the JVM for tests - and so to use command line overrides in tests you must add `Keys.fork in Test := false` in `build.sbt` before you can use them for a test.

**Next:** [Configuring the application secret](#)

## The Application Secret

Play uses a secret key for a number of things, including:

- Signing session cookies and CSRF tokens
- Built in encryption utilities

It is configured in `application.conf`, with the property name `play.crypto.secret`, and defaults to `changeme`. As the default suggests, it should be changed for production.

When started in prod mode, if Play finds that the secret is not set, or if it is set to `changeme`, Play will throw an error.

## Best practices

Anyone that can get access to the secret will be able to generate any session they please, effectively allowing them to log in to your system as any user they please. Hence it is

strongly recommended that you do not check your application secret into source control. Rather, it should be configured on your production server. This means that it is considered bad practice to put the production application secret in `application.conf`.

One way of configuring the application secret on a production server is to pass it as a system property to your start script. For example:

```
/path/to/yourapp/bin/yourapp -Dplay.crypto.secret="QCY?tAnfk?aZ?iwrNwnxIIR6CTf:G3gf:90Latabg@5241AB`R5W:1uDFN];Ik@n"
```

This approach is very simple, and we will use this approach in the Play documentation on running your app in production mode as a reminder that the application secret needs to be set. In some environments however, placing secrets in command line arguments is not considered good practice. There are two ways to address this.

## Environment variables

The first is to place the application secret in an environment variable. In this case, we recommend you place the following configuration in your `application.conf` file:

```
play.crypto.secret="changeme"
play.crypto.secret=${?APPLICATION_SECRET}
```

The second line in that configuration sets the secret to come from an environment variable called `APPLICATION_SECRET` if such an environment variable is set, otherwise, it leaves the secret unchanged from the previous line.

This approach works particularly well for cloud based deployment scenarios, where the normal practice is to set passwords and other secrets via environment variables that can be configured through the API for that cloud provider.

## Production configuration file

Another approach is to create a `production.conf` file that lives on the server, and includes `application.conf`, but also overrides any sensitive configuration, such as the application secret and passwords.

For example:

```
include "application"

play.crypto.secret="QCY?tAnfk?aZ?iwrNwnxIIR6CTf:G3gf:90Latabg@5241AB`R5W:1uDFN];Ik@n"
```

Then you can start Play with:

```
/path/to/yourapp/bin/yourapp -Dconfig.file=/path/to/production.conf
```

# Generating an application secret

Play provides a utility that you can use to generate a new secret.

Run `playGenerateSecret` in the Play console. This will generate a new secret that you can use in your application. For example:

```
[my-first-app] $ playGenerateSecret
[info] Generated new secret: QCYtAnfkaZiwrNwnxIIR6CTfG3gf90Latabg5241ABR5W1uDFNIkn
[success] Total time: 0 s, completed 28/03/2014 2:26:09 PM
```

## Updating the application secret in application.conf

Play also provides a convenient utility for updating the secret in `application.conf`, should you want to have a particular secret configured for development or test servers. This is often useful when you have encrypted data using the application secret, and you want to ensure that the same secret is used every time the application is run in dev mode.

To update the secret in `application.conf`, run `playUpdateSecret` in the Play console:

```
[my-first-app] $ playUpdateSecret
[info] Generated new secret: B4FvQWnTp718vr6AHyvdGlHBGNcvuM4y3jUeRCgXxIwBZlbt
[info] Updating application secret in /Users/jroper/tmp/my-first-app/conf/application.conf
[info] Replacing old application secret: play.crypto.secret="changeme"
[success] Total time: 0 s, completed 28/03/2014 2:36:54 PM
```

Next: [Configuring the JDBC connection pool](#)

## Configuring the JDBC pool.

The Play JDBC datasource is managed by [HikariCP](#).

## Special URLs

Play supports special url format for both MySQL and PostgreSQL:

```
To configure MySQL
db.default.url="mysql://user:password@localhost/database"

To configure PostgreSQL
db.default.url="postgres://user:password@localhost/database"
```

A non-standard port of the database service can be specified:

```
To configure MySQL running in Docker
db.default.url="mysql://user:password@localhost:port/database"

To configure PostgreSQL running in Docker
```

```
db.default.url="postgres://user:password@localhost:port/database"
```

# Reference

In addition to the classical `driver`, `url`, `username`, `password` configuration properties, it also supports additional tuning parameters if you need them.

The `play.db.prototype` configuration from the Play JDBC `reference.conf` is used as the prototype for the configuration for all database connections. The defaults for all the available configuration options can be seen here:

```
play {

 modules {
 enabled += "play.api.db.DBModule"
 enabled += "play.api.db.HikariCPModule"
 }

 # Database configuration
 db {
 # The name of the configuration item from which to read database config.
 # So, if set to db, means that db.default is where the configuration for the
 # database named default is found.
 config = "db"

 # The name of the default database, used when no database name is explicitly
 # specified.
 default = "default"

 # The default connection pool.
 # Valid values are:
 # - default - Use the default connection pool provided by the platform (HikariCP)
 # - hikaricp - Use HikariCP
 # - bonecp - Use BoneCP
 # - A FQCN to a class that implements play.api.db.ConnectionPool
 pool = "default"

 # The prototype for database configuration
 prototype = {

 # The connection pool for this database.
 # Valid values are:
 # - default - Delegate to play.db.pool
 # - hikaricp - Use HikariCP
 # - bonecp - Use BoneCP
 # - A FQCN to a class that implements play.api.db.ConnectionPool
 pool = "default"

 # The database driver
 driver = null

 # The database url
 url = null
 }
 }
}
```

```

The username
username = null

The password
password = null

If non null, binds the JNDI name to this data source to the given JNDI name.
jndiName = null

HikariCP configuration options
hikaricp {

 # The datasource class name, if not using a URL
 dataSourceClassName = null

 # Data source configuration options
 dataSource {
 }

 # Whether autocommit should be used
 autoCommit = true

 # The connection timeout
 connectionTimeout = 30 seconds

 # The idle timeout
 idleTimeout = 10 minutes

 # The max lifetime of a connection
 maxLifetime = 30 minutes

 # If non null, the query that should be used to test connections
 connectionTestQuery = null

 # If non null, sets the minimum number of idle connections to maintain.
 minimumIdle = null

 # The maximum number of connections to make.
 maximumPoolSize = 10

 # If non null, sets the name of the connection pool. Primarily used for stats reporting.
 poolName = null

 # Sets whether or not construction of the pool should fail if the minimum number of connections
 # coludn't be created.
 initializationFailFast = true

 # Sets whether internal queries should be isolated
 isolateInternalQueries = false

 # Sets whether pool suspension is allowed. There is a performance impact to enabling it.
 allowPoolSuspension = false
}

```

```

Sets whether connections should be read only
readOnly = false

Sets whether mbeans should be registered
registerMbeans = false

If non null, sets the catalog that should be used on connections
catalog = null

A SQL statement that will be executed after every new connection creation before adding it to the pool
connectionInitSql = null

If non null, sets the transaction isolation level
transactionIsolation = null

The validation timeout to use
validationTimeout = 5 seconds

If non null, sets the threshold for the amount of time that a connection has been out of the pool before it
is
considered to have leaked
leakDetectionThreshold = null
}

BoneCP configuration options
bonecp {

 # Whether autocommit should be used
 autoCommit = true

 # If non null, the transaction isolation level to use.
 isolation = null

 # If non null, sets the catalog to use
 defaultCatalog = null

 # Whether the database should be treated as read only
 readOnly = false

 # Whether opened statements should be automatically closed
 closeOpenStatements = true

 # The pool partition count
 partitionCount = 1

 # The maximum number of connections per partition
 maxConnectionsPerPartition = 30

 # The minimum number of connections per partition
 minConnectionsPerPartition = 5

 # The increment to acquire connections in
}

```

```

acquireIncrement = 1

The acquire retry attempts
acquireRetryAttempts = 10

The delay to wait before retrying to acquire a connection
acquireRetryDelay = 1 second

The connection timeout
connectionTimeout = 1 second

The idle age to expire connections
idleMaxAge = 10 minutes

The maximum a connection should live for
maxConnectionAge = 1 hour

Whether JMX reporting should be disabled
disableJMX = true

Whether statistics should be kept
statisticsEnabled = false

How frequently idle connections should be tested
idleConnectionTestPeriod = 1 minute

Disable connection tracking
disableConnectionTracking = true

The time limit for executing queries. 0 means no time limit.
queryExecuteTimeLimit = 0

Whether the connection should be reset when closed
resetConnectionOnClose = false

Whether unresolved transactions should be detected
detectUnresolvedTransactions = false

An SQL statement to execute to test if a connection is ok after it is created.
Null turns this feature off.
initSQL = null

An SQL statement to execute to test if a connection is ok before giving it out of the pool.
Null turns this feature off.
connectionTestStatement = null

Whether SQL statements should be logged
logStatements = false
}

}
}

Evolutions configuration

```

```

evolutions {

 # Whether evolutions are enabled
 enabled = true

 # Whether evolution updates should be performed with autocommit or in a manually managed transaction
 autocommit = true

 # Whether locks should be used when apply evolutions. If this is true, a locks table will be created, and will
 # be used to synchronise between multiple Play instances trying to apply evolutions. Set this to true in a
 multi
 # node environment.
 useLocks = false

 # Whether evolutions should be automatically applied. In prod mode, this will only apply ups, in dev mode,
 it will
 # cause both ups and downs to be automatically applied.
 autoApply = false

 # Whether downs should be automatically applied. This must be used in combination with autoApply, and
 only applies
 # to prod mode.
 autoApplyDowns = false

 # Db specific configuration. Should be a map of db names to configuration in the same format as this.
 db {

 }

 }
}

```

**Next:** [Configuring Play's thread pools](#)

# Understanding Play thread pools

Play framework is, from the bottom up, an asynchronous web framework. Streams are handled asynchronously using iteratees. Thread pools in Play are tuned to use fewer threads than in traditional web frameworks, since IO in play-core never blocks.

Because of this, if you plan to write blocking IO code, or code that could potentially do a lot of CPU intensive work, you need to know exactly which thread pool is bearing that workload, and you need to tune it accordingly. Doing blocking IO without taking this into account is likely to result in very poor performance from Play framework, for example, you may see only a few requests per second being handled, while CPU usage sits at 5%. In

comparison, benchmarks on typical development hardware (eg, a MacBook Pro) have shown Play to be able to handle workloads in the hundreds or even thousands of requests per second without a sweat when tuned correctly.

---

# Knowing when you are blocking

The most common place where a typical Play application will block is when it's talking to a database. Unfortunately, none of the major databases provide asynchronous database drivers for the JVM, so for most databases, your only option is to use blocking IO. A notable exception to this is [ReactiveMongo](#), a driver for MongoDB that uses Play's Iteratee library to talk to MongoDB.

Other cases when your code may block include:

- Using REST/WebService APIs through a 3rd party client library (ie, not using Play's asynchronous WS API)
  - Some messaging technologies only provide synchronous APIs to send messages
  - When you open files or sockets directly yourself
  - CPU intensive operations that block by virtue of the fact that they take a long time to execute
- In general, if the API you are using returns `Futures`, it is non-blocking, otherwise it is blocking.

Note that you may be tempted to therefore wrap your blocking code in `Futures`. This does not make it non-blocking, it just means the blocking will happen in a different thread. You still need to make sure that the thread pool that you are using has enough threads to handle the blocking.

In contrast, the following types of IO do not block:

- The Play WS API
- Asynchronous database drivers such as ReactiveMongo
- Sending/receiving messages to/from Akka actors

# Play's thread pools

Play uses a number of different thread pools for different purposes:

- **Netty boss/worker thread pools** - These are used internally by Netty for handling Netty IO. An application's code should never be executed by a thread in these thread pools.
- **Play default thread pool** - This is the thread pool in which all of your application code in Play Framework is executed. It is an Akka dispatcher, and is used by the application `ActorSystem`. It can be configured by configuring Akka, described below.

Note that in Play 2.4 several thread pools were combined together into the Play default thread pool.

# Using the default thread pool

All actions in Play Framework use the default thread pool. When doing certain asynchronous operations, for example, calling `map` or `flatMap` on a future, you may need to provide an implicit execution context to execute the given functions in. An execution context is basically another name for a `ThreadPool`.

In most situations, the appropriate execution context to use will be the **Play default thread pool**. This can be used by importing it into your Scala source file:

```
import play.api.libs.concurrent.Execution.Implicits._
```

```
def someAsyncAction = Action.async {
 import play.api.Play.current
 WS.url("http://www.playframework.com").get().map { response =>
 // This code block is executed in the imported default execution context
 // which happens to be the same thread pool in which the outer block of
 // code in this action will be executed.
 Results.Ok("The response code was " + response.status)
 }
}
```

## Configuring the Play default thread pool

The default thread pool can be configured using standard Akka configuration in `application.conf` under the `akka` namespace. Here is default configuration for Play's thread pool:

```
akka {
 fork-join-executor {
 # Settings this to 1 instead of 3 seems to improve performance.
 parallelism-factor = 1.0

 parallelism-max = 24

 # Setting this to LIFO changes the fork-join-executor
 # to use a stack discipline for task scheduling. This usually
 # improves throughput at the cost of possibly increasing
 # latency and risking task starvation (which should be rare).
 task-peeking-mode = LIFO
 }
}
```

This configuration instructs Akka to create 1 thread per available processor, with a maximum of 24 threads in the pool.

You can also try the default Akka configuration:

```
akka {
 fork-join-executor {
 # The parallelism factor is used to determine thread pool size using the
 # following formula: ceil(available processors * factor). Resulting size
```

```

is then bounded by the parallelism-min and parallelism-max values.
parallelism-factor = 3.0

Min number of threads to cap factor-based parallelism number to
parallelism-min = 8

Max number of threads to cap factor-based parallelism number to
parallelism-max = 64
}
}

```

The full configuration options available to you can be found [here](#).

## Using other thread pools

In certain circumstances, you may wish to dispatch work to other thread pools. This may include CPU heavy work, or IO work, such as database access. To do this, you should first create a `ThreadPool`, this can be done easily in Scala:

```
object Contexts {
 implicit val myExecutionContext: ExecutionContext = Akka.system.dispatchers.lookup("my-context")
}
```

In this case, we are using Akka to create the `ExecutionContext`, but you could also easily create your own `ExecutionContexts` using Java executors, or the Scala fork join thread pool, for example. To configure this Akka execution context, you can add the following configuration to your `application.conf`:

```
my-context {
 fork-join-executor {
 parallelism-factor = 20.0
 parallelism-max = 200
 }
}
```

To use this execution context in Scala, you would simply use the scala `Future` companion object function:

```
Future {
 // Some blocking or expensive code here
}(Contexts.myExecutionContext)
```

or you could just use it implicitly:

```
import Contexts.myExecutionContext

Future {
 // Some blocking or expensive code here
}
```

## Class loaders and thread locals

Class loaders and thread locals need special handling in a multithreaded environment such as a Play program.

## Application class loader

In a Play application the [thread context class loader](#) may not always be able to load application classes. You should explicitly use the application class loader to load classes.

### Java

```
Class myClass = Play.application().classloader().loadClass(myClassName);
```

### Scala

Being explicit about loading classes is most important when running Play in development mode (using `run`) rather than production mode. That's because Play's development mode uses multiple class loaders so that it can support automatic application reloading. Some of Play's threads might be bound to a class loader that only knows about a subset of your application's classes.

In some cases you may not be able to explicitly use the application classloader. This is sometimes the case when using third party libraries. In this case you may need to set the[thread context class loader](#) explicitly before you call the third party code. If you do, remember to restore the context class loader back to its previous value once you've finished calling the third party code.

## Java thread locals

Java code in Play uses a `ThreadLocal` to find out about contextual information such as the current HTTP request. Scala code doesn't need to use `ThreadLocal`s because it can use implicit parameters to pass context instead. `ThreadLocal`s are used in Java so that Java code can access contextual information without needing to pass context parameters everywhere.

Java `ThreadLocal`s, along with the correct context `ClassLoader`, are propagated automatically by `ExecutionContextExecutor` objects provided through the `HttpExecution` class. (An `ExecutionContextExecutor` is both a `ScalaExecutionContext` and a Java `Executor`.) These special `ExecutionContextExecutor` objects are automatically created and used by Java actions and Java `Promise` methods. The default objects wrap the default user thread pool. If you want to do your own threading then you should use the `HttpExecution` class' helper methods to get an `ExecutionContextExecutor` object yourself.

In the example below, a user thread pool is wrapped to create a new `ExecutionContext` that propagates thread locals correctly.

```
import play.libs.HttpExecution;
import scala.concurrent.ExecutionContext;
public Promise<Result> index2() {
```

```
// Wrap an existing thread pool, using the context from the current thread
ExecutionContext myEc = HttpExecution.fromThread(myThreadPool);
return Promise.promise(() -> intensiveComputation(), myEc)
 .map((Integer i) -> ok("Got result: " + i), myEc);
}
```

# Best practices

How you should best divide work in your application between different thread pools greatly depends on the types of work that your application is doing, and the control you want to have over how much of which work can be done in parallel. There is no one size fits all solution to the problem, and the best decision for you will come from understanding the blocking-IO requirements of your application and the implications they have on your thread pools. It may help to do load testing on your application to tune and verify your configuration.

Given the fact that JDBC is blocking thread pools can be sized to the # of connections available to a db pool assuming that the thread pool is used exclusively for database access. Any lesser amount of threads will not consume the number of connections available. Any more threads than the number of connections available could be wasteful given contention for the connections.

Below we outline a few common profiles that people may want to use in Play Framework:

## Pure asynchronous

In this case, you are doing no blocking IO in your application. Since you are never blocking, the default configuration of one thread per processor suits your use case perfectly, so no extra configuration needs to be done. The Play default execution context can be used in all cases.

## Highly synchronous

This profile matches that of a traditional synchronous IO based web framework, such as a Java servlet container. It uses large thread pools to handle blocking IO. It is useful for applications where most actions are doing database synchronous IO calls, such as accessing a database, and you don't want or need control over concurrency for different types of work. This profile is the simplest for handling blocking IO.

In this profile, you would simply use the default execution context everywhere, but configure it to have a very large number of threads in its pool, like so:

```
akka {
 akka.loggers = ["akka.event.slf4j.Slf4jLogger"]
 loglevel = WARNING
```

```

actor {
 default-dispatcher = {
 fork-join-executor {
 parallelism-min = 300
 parallelism-max = 300
 }
 }
}

```

This profile is recommended for Java applications that do synchronous IO, since it is harder in Java to dispatch work to other threads.

Note that we use the same value for `parallelism-min` and `parallelism-max`. The reason is that the number of threads is defined by the following formulas :

```

base-nb-threads = nb-processors * parallelism-factor
parallelism-min <= actual-nb-threads <= parallelism-max

```

So if you don't have enough available processors, you will never be able to reach the `parallelism-max` setting.

## Many specific thread pools

This profile is for when you want to do a lot of synchronous IO, but you also want to control exactly how much of which types of operations your application does at once. In this profile, you would only do non blocking operations in the default execution context, and then dispatch blocking operations to different execution contexts for those specific operations.

In this case, you might create a number of different execution contexts for different types of operations, like this:

```

object Contexts {
 implicit val simpleDbLookups: ExecutionContext = Akka.system.dispatchers.lookup("contexts.simple-db-lookups")
 implicit val expensiveDbLookups: ExecutionContext = Akka.system.dispatchers.lookup("contexts.expensive-db-lookups")
 implicit val dbWriteOperations: ExecutionContext = Akka.system.dispatchers.lookup("contexts.db-write-operations")
 implicit val expensiveCpuOperations: ExecutionContext =
 Akka.system.dispatchers.lookup("contexts.expensive-cpu-operations")
}

```

These might then be configured like so:

```

contexts {
 simple-db-lookups {
 fork-join-executor {
 parallelism-factor = 10.0
 }
 }
}

```

```

 }
}

expensive-db-lookups {
 fork-join-executor {
 parallelism-max = 4
 }
}

db-write-operations {
 fork-join-executor {
 parallelism-factor = 2.0
 }
}

expensive-cpu-operations {
 fork-join-executor {
 parallelism-max = 2
 }
}
}

```

Then in your code, you would create `Futures` and pass the relevant `ExecutionContext` for the type of work that `Future` was doing.

**Note:** The configuration namespace can be chosen freely, as long as it matches the dispatcher ID passed to `Akka.system.dispatchers.lookup`.

## Few specific thread pools

This is a combination between the many specific thread pools and the highly synchronized profile. You would do most simple IO in the default execution context and set the number of threads there to be reasonably high (say 100), but then dispatch certain expensive operations to specific contexts, where you can limit the number of them that are done at one time.

Next: [Configuring logging](#)

# Configuring logging

Play uses [Logback](#) as its logging engine, see the [Logback documentation](#) for details on configuration.

## Default configuration

Play uses the following default configuration in production:

```
<!--
~ Copyright (C) 2009-2015 Typesafe Inc. <http://www.typesafe.com>
```

```

-->
<!-- The default logback configuration that Play uses if no other configuration is provided -->
<configuration>

 <conversionRule conversionWord="coloredLevel" converterClass="play.api.Logger$ColoredLevel" />

 <appender name="FILE" class="ch.qos.logback.core.FileAppender">
 <file>${application.home}/logs/application.log</file>
 <encoder>
 <pattern>%date [%level] from %logger in %thread - %message%n%xException</pattern>
 </encoder>
 </appender>

 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
 <encoder>
 <pattern>%coloredLevel %logger{15} - %message%n%xException{10}</pattern>
 </encoder>
 </appender>

 <appender name="ASYNCFILE" class="ch.qos.logback.classic.AsyncAppender">
 <appender-ref ref="FILE" />
 </appender>

 <appender name="ASYNCSTDOUT" class="ch.qos.logback.classic.AsyncAppender">
 <appender-ref ref="STDOUT" />
 </appender>

 <logger name="play" level="INFO" />
 <logger name="application" level="DEBUG" />

 <!-- Off these ones as they are annoying, and anyway we manage configuration ourself -->
 <logger name="com.avaje.ebean.config.PropertyMapLoader" level="OFF" />
 <logger name="com.avaje.ebeaninternal.server.core.XmlConfigLoader" level="OFF" />
 <logger name="com.avaje.ebeaninternal.server.lib.BackgroundThread" level="OFF" />
 <logger name="com.gargoylesoftware.htmlunit.javascript" level="OFF" />

 <root level="WARN">
 <appender-ref ref="ASYNCFILE" />
 <appender-ref ref="ASYNCSTDOUT" />
 </root>

</configuration>

```

A few things to note about this configuration:

- This specifies a file appender that writes to `logs/application.log`.
- The file logger logs full exception stack traces, while the console logger only logs 10 lines of an exception stack trace.
- Play uses ANSI color codes by default in level messages.
- Play puts both the console and the file logger behind the logback [AsyncAppender](#). For details on the performance implications on this, see this [blog post](#).

# Custom configuration

For any custom configuration, you will need to specify your own Logback configuration file.

## Using a configuration file from project source

You can provide a default logging configuration by providing a file `conf/logback.xml`.

## Using an external configuration file

You can also specify a configuration file via a System property. This is particularly useful for production environments where the configuration file may be managed outside of your application source.

Note: The logging system gives top preference to configuration files specified by system properties, secondly to files in the `conf` directory, and lastly to the default. This allows you to customize your application's logging configuration and still override it for specific environments or developer setups.

### Using `-Dlogger.resource`

Specify a configuration file to be loaded from the classpath:

```
$ start -Dlogger.resource=prod-logger.xml
```

### Using `-Dlogger.file`

Specify a configuration file to be loaded from the file system:

```
$ start -Dlogger.file=/opt/prod/logger.xml
```

## Examples

Here's an example of configuration that uses a rolling file appender, as well as a separate appender for outputting an access log:

```
<configuration>

 <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
 <file>${user.dir}/web/logs/application.log</file>
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
 <!-- Daily rollover with compression -->
 <fileNamePattern>application-log-%d{yyyy-MM-dd}.gz</fileNamePattern>
 <!-- keep 30 days worth of history -->
 <maxHistory>30</maxHistory>
 </rollingPolicy>
 <encoder>
 <pattern>%date{yyyy-MM-dd HH:mm:ss ZZZZ} [%level] from %logger in %thread
- %message%n%xException</pattern>
 </encoder>
 </appender>
```

```

</appender>

<appender name="ACCESS_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
 <file>${user.dir}/web/logs/access.log</file>
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
 <!-- daily rollover with compression -->
 <fileNamePattern>access-log-%d{yyyy-MM-dd}.gz</fileNamePattern>
 <!-- keep 1 week worth of history -->
 <maxHistory>7</maxHistory>
 </rollingPolicy>
 <encoder>
 <pattern>%date{yyyy-MM-dd HH:mm:ss ZZZZ} %message%n</pattern>
 <!-- this quadruples logging throughput -->
 <immediateFlush>false</immediateFlush>
 </encoder>
</appender>

<!-- additivity=false ensures access log data only goes to the access log -->
<logger name="access" level="INFO" additivity="false">
 <appender-ref ref="ACCESS_FILE" />
</logger>

<root level="INFO">
 <appender-ref ref="FILE"/>
</root>

</configuration>

```

This demonstrates a few useful features:

- It uses `RollingFileAppender` which can help manage growing log files.
- It writes log files to a directory external to the application so they aren't affected by upgrades, etc.
- The `FILE` appender uses an expanded message format that can be parsed by third party log analytics providers such as Sumo Logic.
- The `access` logger is routed to a separate log file using the `ACCESS_FILE_APPENDER`.
- All loggers are set to a threshold of `INFO` which is a common choice for production logging.

# Akka logging configuration

Akka has its own logging system which may or may not use Play's underlying logging engine depending on how it is configured.

By default, Akka will ignore Play's logging configuration and print log messages to STDOUT using its own format. You can configure the log level in `application.conf`:

```

akka {
 loglevel="INFO"
}

```

To direct Akka to use Play's logging engine, you'll need to do some careful configuration.

First add the following config in `application.conf`:

```
akka {
 loggers = ["akka.event.slf4j.Slf4jLogger"]
 loglevel="DEBUG"
}
```

A couple things to note:

- Setting `akka.loggers` to `["akka.event.slf4j.Slf4jLogger"]` will cause Akka to use Play's underlying logging engine.
- The `akka.loglevel` property sets the threshold at which Akka will forward log requests to the logging engine but does not control logging output. Once the log requests are forwarded, the Logback configuration controls log levels and appenders as normal. You should set `akka.loglevel` to the lowest threshold that you will use in Logback configuration for your Akka components.

Next, refine your Akka logging settings in your Logback configuration:

```
<!-- Set logging for all Akka library classes to INFO -->
<logger name="akka" level="INFO" />
<!-- Set a specific actor to DEBUG -->
<logger name="actors.MyActor" level="DEBUG" />
```

You may also wish to configure an appender for the Akka loggers that includes useful properties such as thread and actor address.

For more information about configuring Akka's logging, including details on Logback and Slf4j integration, see the [Akka documentation](#).

**Next:** [Configuring Play filters](#)

# Configuring gzip encoding

Play provides a gzip filter that can be used to gzip responses.

## Enabling the gzip filter

To enable the gzip filter, add the Play filters project to your `libraryDependencies` in `build.sbt`:

```
libraryDependencies += filters
```

Now add the gzip filter to your filters, which is typically done by creating a `Filters` class in the root of your project:

## Scala

```
import javax.inject.Inject

import play.api.http.HttpFilters
import play.filters.gzip.GzipFilter

class Filters @Inject()(gzipFilter: GzipFilter) extends HttpFilters {
 def filters = Seq(gzipFilter)
}
```

## Java

The `Filters` class can either be in the root package, or if it has another name or is in another package, needs to be configured

using `play.http.filters` in `application.conf`:

```
play.http.filters = "filters.MyFilters"
```

# Configuring the gzip filter

The gzip filter supports a small number of tuning configuration options, which can be configured from `application.conf`. To see the available configuration options, see the Play filters [reference.conf](#).

# Controlling which responses are gzipped

To control which responses are and aren't implemented, use the `shouldGzip` parameter, which accepts a function of a request header and a response header to a boolean.

For example, the code below only gzips HTML responses:

```
new GzipFilter(shouldGzip = (request, response) =>
 response.headers.get("Content-Type").exists(_.startsWith("text/html")))
```

Next: [Configuring security headers](#)

# Configuring Security Headers

Play provides a security headers filter that can be used to configure some default headers in the HTTP response to mitigate security issues and provide an extra level of defense for new applications.

# Enabling the security headers filter

To enable the security headers filter, add the Play filters project to your `libraryDependencies` in `build.sbt`:

`libraryDependencies += filters`

Now add the security headers filter to your filters, which is typically done by creating a `Filters` class in the root of your project:

## Scala

```
import javax.inject.Inject

import play.api.http.HttpFilters
import play.filters.headers.SecurityHeadersFilter

class Filters @Inject()(securityHeadersFilter: SecurityHeadersFilter) extends HttpFilters {
 def filters = Seq(securityHeadersFilter)
}
```

## Java

The `Filters` class can either be in the root package, or if it has another name or is in another package, needs to be configured

using `play.http.filters` in `application.conf`:  
`play.http.filters = "filters.MyFilters"`

# Configuring the security headers

Scaladoc is available in the `play.filters.headers` package.

The filter will set headers in the HTTP response automatically. The settings can be configured through the following settings in `application.conf`

- `play.filters.headers.frameOptions` - sets [X-Frame-Options](#), “DENY” by default.
- `play.filters.headers.xssProtection` - sets [X-XSS-Protection](#), “1; mode=block” by default.
- `play.filters.headers.contentTypeOptions` - sets [X-Content-Type-Options](#), “nosniff” by default.

- `play.filters.headers.permittedCrossDomainPolicies` - sets [X-Permitted-Cross-Domain-Policies](#), “master-only” by default.
  - `play.filters.headers.contentSecurityPolicy` - sets [Content-Security-Policy](#), “default-src ‘self’” by default.  
Any of the headers can be disabled by setting a configuration value of `null`, for example:  
`play.filters.headers.frameOptions = null`  
For a full listing of configuration options, see the Play filters [reference.conf](#).
- Next: [Configuring CORS](#)

# Cross-Origin Resource Sharing

Play provides a filter that implements Cross-Origin Resource Sharing (CORS).

CORS is a protocol that allows web applications to make requests from the browser across different domains. A full specification can be found [here](#).

## Enabling the CORS filter

To enable the CORS filter, add the Play filters project to your `libraryDependencies` in `build.sbt`:

`libraryDependencies += filters`

Now add the CORS filter to your filters, which is typically done by creating a `Filters` class in the root of your project:

### Scala

```
import javax.inject.Inject

import play.api.http.HttpFilters
import play.filters.cors.CORSFilter

class Filters @Inject() (corsFilter: CORSFilter) extends HttpFilters {
 def filters = Seq(corsFilter)
}
```

### Java

# Configuring the CORS filter

The filter can be configured from `application.conf`. For a full listing of configuration options, see the Play filters `reference.conf`.

The available options include:

- `play.filters.cors.pathPrefixes` - filter paths by a whitelist of path prefixes
- `play.filters.cors.allowedOrigins` - allow only requests with origins from a whitelist (by default all origins are allowed)
- `play.filters.cors.allowedHttpMethods` - allow only HTTP methods from a whitelist for preflight requests (by default all methods are allowed)
- `play.filters.cors.allowedHttpHeaders` - allow only HTTP headers from a whitelist for preflight requests (by default all headers are allowed)
- `play.filters.cors.exposedHeaders` - set custom HTTP headers to be exposed in the response (by default no headers are exposed)
- `play.filters.cors.supportsCredentials` - disable/enable support for credentials (by default credentials support is enabled)
- `play.filters.cors.preflightMaxAge` - set how long the results of a preflight request can be cached in a preflight result cache (by default 1 hour)

For example:

```
play.filters.cors {
 pathPrefixes = ["/some/path", ...]
 allowedOrigins = ["http://www.example.com", ...]
 allowedHttpMethods = ["GET", "POST"]
 allowedHttpHeaders = ["Accept"]
 preflightMaxAge = 3 days
}
```

Next: [Configuring WS SSL](#)

# Configuring WS SSL

Play WS allows you to set up HTTPS completely from a configuration file, without the need to write code. It does this by layering the Java Secure Socket Extension (JSSE) with a configuration layer and with reasonable defaults.

JDK 1.8 contains an implementation of JSSE which is significantly more advanced than previous versions, and should be used if security is a priority.

---

# Table of Contents

- Quick Start to WS SSL
  - Generating X.509 Certificates
  - Configuring Trust Stores and Key Stores
  - Configuring Protocols
  - Configuring Cipher Suites
  - Configuring Certificate Validation
  - Configuring Certificate Revocation
  - Configuring Hostname Verification
  - Example Configurations
  - Using the Default SSLContext
  - Debugging SSL Connections
  - Loose Options
  - Testing SSL
- 

## Further Reading

JSSE is a complex product. For convenience, the JSSE materials are provided here:

JDK 1.8:

- JSSE Reference Guide
- JSSE Crypto Spec
- SunJSSE Providers
- PKI Programmer's Guide
- keytool

**Next:** [Quick Start to WS SSL](#)

## Quick Start to WS SSL

This section is for people who need to connect to a remote web service over HTTPS, and don't want to read through the entire manual. If you need to set up a web service or configure client authentication, please proceed to the [next section](#).

## Connecting to a Remote Server over HTTPS

If the remote server is using a certificate that is signed by a well known certificate authority, then WS should work out of the box without any additional configuration. You're done!

If the web service is not using a well known certificate authority, then it is using either a private CA or a self-signed certificate. You can determine this easily by using curl:

```
curl https://financialcryptography.com # uses cacert.org as a CA
```

If you receive the following error:

```
curl: (60) SSL certificate problem: Invalid certificate chain
More details here: http://curl.haxx.se/docs/sslcerts.html
```

curl performs SSL certificate verification **by default, using a "bundle"** of **Certificate Authority (CA)** **public** keys (CA certs). **If the default** bundle file isn't adequate, you can specify an alternate file using the --cacert option.

If this HTTPS server uses a certificate signed by a CA represented in the bundle, the certificate verification probably failed due to a problem with the certificate (it might be expired, or the name might not match the domain name in the URL).

If you'd like to turn off curl's verification of the certificate, use the -k (or --insecure) option.

Then you have to obtain the CA's certificate, and add it to the trust store.

## Obtain the Root CA Certificate

Ideally this should be done out of band: the owner of the web service should provide you with the root CA certificate directly, in a way that can't be faked, preferably in person.

In the case where there is no communication (and this is **not recommended**), you can sometimes get the root CA certificate directly from the certificate chain, using `keytool`

```
from JDK 1.8:
```

```
keytool -printcert -sslserver playframework.com
```

which returns #2 as the root certificate:

```
Certificate #2
```

```
=====
Owner: CN=GlobalSign Root CA, OU=Root CA, O=GlobalSign nv-sa, C=BE
Issuer: CN=GlobalSign Root CA, OU=Root CA, O=GlobalSign nv-sa, C=BE
```

To get the certificate chain in an exportable format, use the -rfc option:

```
keytool -printcert -sslserver playframework.com -rfc
```

which will return a series of certificates in PEM format:

```
--BEGIN CERTIFICATE--
...
--END CERTIFICATE--
```

which can be copied and pasted into a file. The very last certificate in the chain will be the root CA certificate.

**NOTE:** Not all websites will include the root CA certificate. You should decode the certificate with keytool or with [certificate decoder](#) to ensure you have the right certificate.

## Point the trust manager at the PEM file

Add the following into `conf/application.conf`, specifying `PEM` format specifically:

```
play.ws.ssl {
 trustManager = {
 stores = [
 { type = "PEM", path = "/path/to/cert/globalsign.crt" }
]
 }
}
```

This will tell the trust manager to ignore the default `cacerts` store of certificates, and only use your custom CA certificate.

After that, WS will be configured, and you can test that your connection works with:

```
WS.url("https://example.com").get()
```

You can see more examples on the [example configurations](#) page.

Next: [Generating X.509 Certificates](#)

# Generating X.509 Certificates

## X.509 Certificates

Public key certificates are a solution to the problem of identity. Encryption alone is enough to set up a secure connection, but there's no guarantee that you are talking to the server that you think you are talking to. Without some means to verify the identity of a remote server, an attacker could still present itself as the remote server and then forward the secure connection onto the remote server. Public key certificates solve this problem.

The best way to think about public key certificates is as a passport system. Certificates are used to establish information about the bearer of that information in a way that is difficult to forge. This is why certificate verification is so important: accepting **any**certificate means that even an attacker's certificate will be blindly accepted.

---

## Using Keytool

Use the keytool version that comes with JDK 8:

- 1.8

The examples below use keytool 1.8 for marking a certificate for CA usage or for a hostname.

---

## Generating a random password

Create a random password using pwgen (`brew install pwgen` if you're on a Mac):  
export PW=`pwgen -Bs 10 1`  
echo \$PW > password

## Server Configuration

You will need a server with a DNS hostname assigned, for hostname verification. In this example, we assume the hostname is `example.com`.

### Generating a server CA

The first step is to create a certificate authority that will sign the `example.com` certificate. The root CA certificate has a couple of additional attributes (`ca:true, keyCertSign`) that mark it explicitly as a CA certificate, and will be kept in a trust store.

```
export PW=`cat password`

Create a self signed key pair root CA certificate.
keytool -genkeypair -v \
-alias exampleca \
-dname "CN=exampleCA, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US" \
```

```

-keystore exampleca.jks \
-keypass:env PW \
-storepass:env PW \
-keyalg RSA \
-keyszie 4096 \
-ext KeyUsage:critical="keyCertSign" \
-ext BasicConstraints:critical="ca:true" \
-validity 9999

Export the exampleCA public certificate as exampleca.crt so that it can be used in trust stores.
keytool -export -v \
-alias exampleca \
-file exampleca.crt \
-keypass:env PW \
-storepass:env PW \
-keystore exampleca.jks \
-rfc

```

## Generating example.com certificates

The example.com certificate is presented by the `example.com` server in the handshake.  
export PW=`cat password`

```

Create a server certificate, tied to example.com
keytool -genkeypair -v \
-alias example.com \
-dname "CN=example.com, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US" \
-keystore example.com.jks \
-keypass:env PW \
-storepass:env PW \
-keyalg RSA \
-keyszie 2048 \
-validity 385

Create a certificate signing request for example.com
keytool -certreq -v \
-alias example.com \
-keypass:env PW \
-storepass:env PW \
-keystore example.com.jks \
-file example.com.csr

Tell exampleCA to sign the example.com certificate. Note the extension is on the request, not the
original certificate.
Technically, keyUsage should be digitalSignature for DHE or ECDHE, keyEncipherment for RSA.
keytool -gencert -v \
-alias exampleca \
-keypass:env PW \
-storepass:env PW \
-keystore exampleca.jks \
-infile example.com.csr \
-outfile example.com.crt \
-ext KeyUsage:critical="digitalSignature,keyEncipherment" \

```

```

-ext EKU="serverAuth" \
-ext SAN="DNS:example.com" \
-rfc

Tell example.com.jks it can trust exampleca as a signer.
keytool -import -v \
 -alias exampleca \
 -file exampleca.crt \
 -keystore example.com.jks \
 -storetype JKS \
 -storepass:env PW << EOF
yes
EOF

Import the signed certificate back into example.com.jks
keytool -import -v \
 -alias example.com \
 -file example.com.crt \
 -keystore example.com.jks \
 -storetype JKS \
 -storepass:env PW

List out the contents of example.com.jks just to confirm it.
If you are using Play as a TLS termination point, this is the key store you should present as the server.
keytool -list -v \
 -keystore example.com.jks \
 -storepass:env PW

```

You should see:

```

Alias name: example.com
Creation date: ...
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=example.com, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US
Issuer: CN=exampleCA, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US

```

**NOTE:** Also see the [Configuring HTTPS](#) section for more information.

## Configuring example.com certificates in Nginx

If example.com does not use Java as a TLS termination point, and you are using nginx, you may need to export the certificates in PEM format.

Unfortunately, keytool does not export private key information, so openssl must be installed to pull private keys.

```

export PW=`cat password`

Export example.com's public certificate for use with nginx.
keytool -export -v \
 -alias example.com \

```

```

-file example.com.crt \
-keypass:env PW \
-storepass:env PW \
-keystore example.com.jks \
-rfc

Create a PKCS#12 keystore containing the public and private keys.
keytool -importkeystore -v \
-sralias example.com \
-srckeystore example.com.jks \
-srcstoretype jks \
-srcstorepass:env PW \
-destkeystore example.com.p12 \
-destkeypass:env PW \
-deststorepass:env PW \
-deststoretype PKCS12

Export the example.com private key for use in nginx. Note this requires the use of OpenSSL.
openssl pkcs12 \
-nocerts \
-nodes \
-passout env:PW \
-passin env:PW \
-in example.com.p12 \
-out example.com.key

Clean up.
rm example.com.p12

```

Now that you have both `example.com.crt` (the public key certificate) and `example.com.key` (the private key), you can set up an HTTPS server.

For example, to use the keys in nginx, you would set the following in `nginx.conf`:

```

ssl_certificate /etc/nginx/certs/example.com.crt;
ssl_certificate_key /etc/nginx/certs/example.com.key;

```

If you are using client authentication (covered in [Client Configuration](#) below), you will also need to add:

```

ssl_client_certificate /etc/nginx/certs/clientca.crt;
ssl_verify_client on;

```

You can check the certificate is what you expect by checking the server:

```
keytool -printcert -sslserver example.com
```

**NOTE:** Also see the [Setting up a front end HTTP server](#) section for more information.

# Client Configuration

There are two parts to setting up a client – configuring a trust store, and configuring client authentication.

# Configuring a Trust Store

Any clients need to see that the server's example.com certificate is trusted, but don't need to see the private key. Generate a trust store which contains only the certificate and hand that out to clients. Many java clients prefer to have the trust store in JKS format.

```
export PW=`cat password`

Create a JKS keystore that trusts the example CA, with the default password.
keytool -import -v \
-alias exampleca \
-file exampleca.crt \
-keypass:env PW \
-storepass changeit \
-keystore exampletrust.jks << EOF
yes
EOF

List out the details of the store password.
keytool -list -v \
-keystore exampletrust.jks \
-storepass changeit
```

You should see a `trustedCertEntry` for exampleca:

```
Alias name: exampleca
Creation date: ...
Entry type: trustedCertEntry
```

Owner: CN=exampleCA, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US  
Issuer: CN=exampleCA, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US

The `exampletrust.jks` store will be used in the TrustManager.

```
play.ws.ssl {
 trustManager = {
 stores = [
 { path = "/Users/wsargent/work/ssltest/conf/exampletrust.jks" }
]
 }
}
```

**NOTE:** Also see the [Configuring Key Stores and Trust Stores](#) section for more information.

## Configure Client Authentication

Client authentication can be obscure and poorly documented, but it relies on the following steps:

1. The server asks for a client certificate, presenting a CA that it expects a client certificate to be signed with. In this case, `CN=clientCA` (see the [debug example](#)).
2. The client looks in the KeyManager for a certificate which is signed by `clientCA`, using `chooseClientAlias` and `certRequest.getAuthorities`.
3. The KeyManager will return the `client` certificate to the server.
4. The server will do an additional ClientKeyExchange in the handshake.

The steps to create a client CA and a signed client certificate are broadly similar to the server certificate generation, but for convenience are presented in a single script:

```
export PW=`cat password`

Create a self signed certificate & private key to create a root certificate authority.
keytool -genkeypair -v \
-alias clientca \
-keystore client.jks \
-dname "CN=clientca, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US" \
-keypass:env PW \
-storepass:env PW \
-keyalg RSA \
-keysize 4096 \
-ext KeyUsage:critical="keyCertSign" \
-ext BasicConstraints:critical="ca:true" \
-validity 9999

Create another key pair that will act as the client.
keytool -genkeypair -v \
-alias client \
-keystore client.jks \
-dname "CN=client, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US" \
-keypass:env PW \
-storepass:env PW \
-keyalg RSA \
-keysize 2048

Create a certificate signing request from the client certificate.
keytool -certreq -v \
-alias client \
-keypass:env PW \
-storepass:env PW \
-keystore client.jks \
-file client.csr

Make clientCA create a certificate chain saying that client is signed by clientCA.
keytool -gencert -v \
-alias clientca \
-keypass:env PW \
-storepass:env PW \
-keystore client.jks \
-infile client.csr \
-outfile client.crt \
-ext EKU="clientAuth" \
-rfc

Export the client-ca certificate from the keystore. This goes to nginx under "ssl_client_certificate"
and is presented in the CertificateRequest.
keytool -export -v \
-alias clientca \
-file clientca.crt \
-storepass:env PW \
-keypass:env PW
```

```

-keystore client.jks \
-rfc

Import the signed certificate back into client.jks. This is important, as JSSE won't send a client
certificate if it can't find one signed by the client-ca presented in the CertificateRequest.
keytool -import -v \
-alias client \
-file client.crt \
-keystore client.jks \
-storetype JKS \
-storepass:env PW

Export the client CA's certificate and private key to pkcs12, so it's safe.
keytool -importkeystore -v \
-sralias clientca \
-srckeystore client.jks \
-srcstorepass:env PW \
-destkeystore clientca.p12 \
-deststorepass:env PW \
-deststoretype PKCS12

Import the client CA's public certificate into a JKS store for Play Server to read. We don't use
the PKCS12 because it's got the CA private key and we don't want that.
keytool -import -v \
-alias clientca \
-file clientca.crt \
-keystore clientca.jks \
-storepass:env PW << EOF
yes
EOF

Then, strip out the client CA alias from client.jks, just leaving the signed certificate.
keytool -delete -v \
-alias clientca \
-storepass:env PW \
-keystore client.jks

List out the contents of client.jks just to confirm it.
keytool -list -v \
-keystore client.jks \
-storepass:env PW

```

There should be one alias `client`, looking like the following:

Your keystore contains 1 entry

```

Alias name: client
Creation date: ...
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=client, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US
Issuer: CN=clientCA, OU=Example Org, O=Example Company, L=San Francisco, ST=California, C=US
And put client.jks in the key manager:
play.ws.ssl {

```

```
keyManager = {
 stores = [
 { type = "JKS", path = "conf/client.jks", password = $PW }
]
}
```

**NOTE:** Also see the [Configuring Key Stores and Trust Stores](#) section for more information.

## Certificate Management Tools

If you want to examine certificates in a graphical tool than a command line tool, you can use [Keystore Explorer](#) or [xca](#). [Keystore Explorer](#) is especially convenient as it recognizes JKS format. It works better as a manual installation, and requires some tweaking to the export policy.

If you want to use a command line tool with more flexibility than keytool, try [java-keyutil](#), which understands multi-part PEM formatted certificates and JKS.

## Certificate Settings

### Secure

If you want the best security, consider using [ECDSA](#) as the signature algorithm (in keytool, this would be `-sigalg EC`). ECDSA is also known as “ECC SSL Certificate”.

### Compatible

For compatibility with older systems, use RSA with 2048 bit keys and SHA256 as the signature algorithm. If you are creating your own CA certificate, use 4096 bits for the root.

## Further Reading

- [JSSE Reference Guide To Creating KeyStores](#)
- [Java PKI Programmer’s Guide](#)
- [Fixing X.509 Certificates](#)

**Next:** [Configuring Trust Stores and Key Stores](#)

# Configuring Trust Stores and Key Stores

Trust stores and key stores contain X.509 certificates. Those certificates contain public (or private) keys, and are organized and managed under either a TrustManager or a KeyManager, respectively.

If you need to generate X.509 certificates, please see [Certificate Generation](#) for more information.

## Configuring a Trust Manager

A [trust manager](#) is used to keep trust anchors: these are root certificates which have been issued by certificate authorities. It determines whether the remote authentication credentials (and thus the connection) should be trusted. As an HTTPS client, the vast majority of requests will use only a trust manager.

If you do not configure it at all, WS uses the default trust manager, which points to the `cacerts` key store in  `${java.home}/lib/security/cacerts`. If you configure a trust manager explicitly, it will override the default settings and the `cacerts` store will not be included.

If you wish to use the default trust store and add another store containing certificates, you can define multiple stores in your trust manager. The [CompositeX509TrustManager](#) will try each in order until it finds a match:

```
play.ws.ssl {
 trustManager = {
 stores = [
 { path: ${store.directory}/truststore.jks, type: "JKS" } # Added trust store
 { path: ${java.home}/lib/security/cacerts, password = "changeit" } # Default trust store
]
 }
}
```

**NOTE:** Trust stores should only contain CA certificates with public keys, usually JKS or PEM. PKCS12 format is supported, but PKCS12 should not contain private keys in a trust store, as noted in the [reference guide](#).

## Configuring a Key Manager

A [key manager](#) is used to present keys for a remote host. Key managers are typically only used for client authentication (also known as mutual TLS).

The [CompositeX509KeyManager](#) may contain multiple stores in the same manner as the trust manager.

```
play.ws.ssl {
 keyManager = {
 stores = [
 {
 type: "pkcs12",
 path: "keystore.p12",
 password: "password1"
 }
]
 }
}
```

```
 },
]
}
}
```

**NOTE:** A key store that holds private keys should use PKCS12 format, as indicated in the [reference guide](#).

## Configuring a Store

A store corresponds to a `KeyStore` object, which is used for both trust stores and key stores. Stores may have a `type` – `PKCS12`, `JKS` or `PEM` (aka Base64 encoded DER certificate) – and may have an associated password.

Stores must have either a `path` or a `data` attribute. The `path` attribute must be a file system path.

```
{ type: "PKCS12", path: "/private/keystore.p12" }
```

The `data` attribute must contain a string of PEM encoded certificates.

```
{
 type: "PEM", data = """
-----BEGIN CERTIFICATE-----
...certificate data
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
...certificate data
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
...certificate data
-----END CERTIFICATE-----
"""
}
```

## Debugging

To debug the key manager / trust manager, set the following flags:

```
play.ws.ssl.debug = {
 ssl = true
 trustmanager = true
 keymanager = true
}
```

## Further Reading

In most cases, you will not need to do extensive configuration once the certificates are installed. If you are having difficulty with configuration, the following blog posts may be useful:

- [Key Management](#)
- [Java 2-way TLS/SSL \(Client Certificates\) and PKCS12 vs JKS KeyStores](#)
- [HTTPS with Client Certificates on Android](#)

Next: [Configuring Protocols](#)

# Configuring Protocols

By default, WS SSL will use the most secure version of the TLS protocol available in the JVM.

- On JDK 1.7 and later, the default protocol is “TLSv1.2”.
- On JDK 1.6, the default protocol is “TLSv1”.

The full protocol list in JSSE is available in the [Standard Algorithm Name Documentation](#).

## Defining the default protocol

If you want to define a different [default protocol](#), you can set it specifically in the client:

```
Passed into SSLContext.getInstance()
play.ws.ssl.protocol = "TLSv1.2"
```

If you want to define the list of enabled protocols, you can do so explicitly:

```
passed into sslContext.getDefaultParameters().setEnabledProtocols()
play.ws.ssl.enabledProtocols = [
 "TLSv1.2",
 "TLSv1.1",
 "TLSv1"
]
```

If you are on JDK 1.8, you can also set the `jdk.tls.client.protocols` system property to enable client protocols globally.

WS recognizes “SSLv3”, “SSLv2” and “SSLv2Hello” as weak protocols with a number of [security issues](#), and will throw an exception if they are in the `play.ws.ssl.enabledProtocols` list. Virtually all servers support `TLSv1`, so there is no advantage in using these older protocols.

## Debugging

The debug options for configuring protocol are:

```
play.ws.ssl.debug = {
 ssl = true
 sslctx = true
```

```
handshake = true
verbose = true
data = true
}
```

Next: [Configuring Cipher Suites](#)

# Configuring Cipher Suites

A [cipher suite](#) is really four different ciphers in one, describing the key exchange, bulk encryption, message authentication and random number function. There is [no official naming convention](#) of cipher suites, but most cipher suites are described in order – for example, “TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA” uses DHE for key exchange, RSA for server certificate authentication, 256-bit key AES in CBC mode for the stream cipher, and SHA for the message authentication.

## Configuring Enabled Ciphers

The list of cipher suites has changed considerably between 1.6, 1.7 and 1.8.

In 1.7 and 1.8, the default [out of the box](#) cipher suite list is used.

In 1.6, the out of the box list is [out of order](#), with some weaker cipher suites configured in front of stronger ones, and contains a number of ciphers that are now considered weak. As such, the default list of enabled cipher suites is as follows:

```
"TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
"TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
"TLS_DHE_DSS_WITH_AES_128_CBC_SHA",
"TLS_RSA_WITH_AES_256_CBC_SHA",
"TLS_RSA_WITH_AES_128_CBC_SHA",
"SSL_RSA_WITH_RC4_128_SHA",
"SSL_RSA_WITH_RC4_128_MD5",
"TLS_EMPTY_RENEGOTIATION_INFO_SCSV" // per RFC 5746
```

The list of cipher suites can be configured manually using the `play.ws.ssl.enabledCiphers` setting:

```
play.ws.ssl.enabledCiphers = [
 "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256"
]
```

This can be useful to enable perfect forward security, for example, as only DHE and ECDHE cipher suites enable PFS.

---

# Recommendation: increase the DHE key size

Diffie Hellman has been in the news recently because it offers perfect forward secrecy. However, in 1.6 and 1.7, the server handshake of DHE is set to 1024 at most, which is considered weak and can be compromised by attackers.

If you have JDK 1.8, setting the system property `-Djdk.tls.ephemeralDHKeySize=2048`

is recommended to ensure stronger keysizes in the handshake. Please see [Customizing Size of Ephemeral Diffie-Hellman Keys](#).

---

# Recommendation: Use Ciphers with Perfect Forward Secrecy

As per the [Recommendations for Secure Use of TLS and DTLS](#), the following cipher suites are recommended:

```
play.ws.ssl.enabledCiphers = [
 "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256",
 "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
 "TLS_DHE_RSA_WITH_AES_256_GCM_SHA384",
 "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
]
```

Some of these ciphers are only available in JDK 1.8.

---

# Disabling Weak Ciphers and Weak Key Sizes Globally

The `jdk.tls.disabledAlgorithms` can be used to prevent weak ciphers, and can also be used to prevent [small key sizes](#) from being used in a handshake. This is a [useful feature](#) that is only available in Oracle JDK 1.7 and later.

The official documentation for disabled algorithms is in the [JSSE Reference Guide](#).

For TLS, the code will match the first part of the cipher suite after the protocol, i.e.

`TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384` has ECDHE as the relevant cipher. The parameter names to use for the disabled algorithms are not obvious, but are listed in the [Providers documentation](#) and can be seen in the [source code](#).

To enable `jdk.tls.disabledAlgorithms` or `jdk.certpath.disabledAlgorithms` (which looks at signature algorithms and weak keys in X.509 certificates) you must create a properties file:

```
disabledAlgorithms.properties
jdk.tls.disabledAlgorithms=EC keySize < 160, RSA keySize < 2048, DSA keySize < 2048
jdk.certpath.disabledAlgorithms=MD2, MD4, MD5, EC keySize < 160, RSA keySize < 2048, DSA keySize < 2048
```

And then start up the JVM with `java.security.properties`:

```
java -Djava.security.properties=disabledAlgorithms.properties
```

## Debugging

To debug ciphers and weak keys, turn on the following debug settings:

```
play.ws.ssl.debug = {
 ssl = true
 handshake = true
 verbose = true
 data = true
}
```

Next: [Configuring Certificate Validation](#)

Next: [Configuring Certificate Validation](#)

# Configuring Certificate Validation

In an SSL connection, the identity of the remote server is verified using an X.509 certificate which has been signed by a certificate authority.

The JSSE implementation of X.509 certificates is defined in the [PKI Programmer's Guide](#). Some X.509 certificates that are used by servers are old, and are using signatures that can be forged by an attacker. Because of this, it may not be possible to verify the identity of the server if that signature algorithm is being used. Fortunately, this is rare – over 95% of trusted leaf certificates and 95% of trusted signing certificates use [NIST recommended key sizes](#).

WS automatically disables weak signature algorithms and weak keys for you, according to the [current standards](#).

This feature is similar to [jdk.certpath.disabledAlgorithms](#), but is specific to the WS client and can be set dynamically, whereas jdk.certpath.disabledAlgorithms is global across the JVM, must be set via a security property, and is only available in JDK 1.7 and later.

You can override this to your tastes, but it is recommended to be at least as strict as the defaults. The appropriate signature names can be looked up in the [Providers Documentation](#).

# Disabling Certificates with Weak Signature Algorithms

The default list of disabled signature algorithms is defined below:

```
play.ws.ssl.disabledSignatureAlgorithms = "MD2, MD4, MD5"
```

MD5 is disabled, based on the proven [collision attack](#) and the Mozilla recommendations:

MD5 certificates may be compromised when attackers can create a fake cert that hashes to the same value as one with a legitimate signature, and is hence trusted. Mozilla can mitigate this potential vulnerability by turning off support for MD5-based signatures. The MD5 root certificates don't necessarily need to be removed from NSS, because the signatures of root certificates are not validated (roots are self-signed). Disabling MD5 will impact intermediate and end entity certificates, where the signatures are validated.

The relevant CAs have confirmed that they stopped issuing MD5 certificates. However, there are still many end entity certificates that would be impacted if support for MD5-based signatures was turned off in 2010. Therefore, we are hoping to give the affected CAs time to react, and are proposing the date of June 30, 2011 for turning off support for MD5-based signatures. The relevant CAs are aware that Mozilla will turn off MD5 support earlier if needed.

SHA-1 is considered weak, and new certificates should use digest algorithms from the [SHA-2 family](#). However, old certificates should still be considered valid.

# Disabling Certificates With Weak Key Sizes

WS defines the default list of weak key sizes as follows:

```
play.ws.ssl.disabledKeyAlgorithms = "DHE keySize < 2048, ECDH keySize < 2048, ECDHE keySize < 2048,
RSA keySize < 2048, DSA keySize < 2048, EC keySize < 224"
```

These settings are based in part on [keylength.com](#), and in part on the Mozilla recommendations:

The NIST recommendation is to discontinue 1024-bit RSA certificates by December 31, 2010. Therefore, CAs have been advised that they should not sign any more certificates under their 1024-bit roots by the end of this year.

The date for disabling/removing 1024-bit root certificates will be dependent on the state of the art in public key cryptography, but under no circumstances should any party expect continued support for this modulus size past December 31, 2013. As mentioned above, this date could get moved up substantially if new attacks are discovered. We recommend all parties involved in secure transactions on the web move away from 1024-bit moduli as soon as possible.

**NOTE:** because weak key sizes also apply to root certificates (which is not included in the certificate chain available to the PKIX certpath checker included in JSSE), setting this option will check the accepted issuers in any configured trustmanagers and keymanagers, including the default.

Over 95% of trusted leaf certificates and 95% of trusted signing certificates use [NIST recommended key sizes](#), so this is considered a safe default.

## Disabling Weak Certificates Globally

To disable signature algorithms and weak key sizes globally across the JVM, use the `jdk.certpath.disabledAlgorithms` security property. Setting security properties is covered in more depth in [Configuring Cipher Suites](#) section.

**NOTE** if configured, the `jdk.certpath.disabledAlgorithms` property should contain the settings from both `disabledKeyAlgorithms` and `disabledSignatureAlgorithms`.

## Debugging Certificate Validation

To see more details on certificate validation, set the following debug configuration:

```
play.ws.ssl.debug.certpath = true
```

The undocumented setting `-Djava.security.debug=x509` may also be helpful.

## Further Reading

- [Dates for Phasing out MD5-based signatures and 1024-bit moduli](#)
- [Fixing X.509 Certificates](#)

Next: [Configuring Certificate Revocation](#)

# Configuring Certificate Revocation

Certificate Revocation in JSSE can be done through two means: certificate revocation lists (CRLs) and OCSP.

Certificate Revocation can be very useful in situations where a server's private keys are compromised, as in the case of [Heartbleed](#).

Certificate Revocation is disabled by default in JSSE. It is defined in two places:

- [PKI Programmer's Guide, Appendix C](#)
- [Enable OCSP Checking](#)

To enable OCSP, you must set the following system properties on the command line:

```
java -Dcom.sun.security.enableCRLDP=true -Dcom.sun.net.ssl.checkRevocation=true
```

After doing the above, you can enable certificate revocation in the client:

```
play.ws.ssl.checkRevocation = true
Setting checkRevocation will set the internal ocsp.enable security property
automatically:
java.security.Security.setProperty("ocsp.enable", "true")
```

And this will set OCSP checking when making HTTPS requests.

NOTE: Enabling OCSP requires a round trip to the OCSP responder. This adds a notable overhead on HTTPS calls, and can make calls up to [33% slower](#). The mitigation technique, OCSP stapling, is not supported in JSSE.

Or, if you wish to use a static CRL list, you can define a list of URLs:

```
play.ws.ssl.revocationLists = ["http://example.com/crl"]
```

## Debugging

To test certificate revocation is enabled, set the following options:

```
play.ws.ssl.debug = {
 certpath = true
```

```
ocsp = true
}
```

And you should see something like the following output:

```
certpath: -Using checker7 ... [sun.security.provider.certpath.RevocationChecker]
certpath: connecting to OCSP service at: http://gtssl2-ocsp.geotrust.com
certpath: OCSP response status: SUCCESSFUL
certpath: OCSP response type: basic
certpath: Responder's name: CN=GeoTrust SSL CA - G2 OCSP Responder, O=GeoTrust Inc., C=US
certpath: OCSP response produced at: Wed Mar 19 13:57:32 PDT 2014
certpath: OCSP number of SingleResponses: 1
certpath: OCSP response cert #1: CN=GeoTrust SSL CA - G2 OCSP Responder, O=GeoTrust Inc., C=US
certpath: Status of certificate (with serial number 159761413677206476752317239691621661939) is: GOOD
certpath: Responder's certificate includes the extension id-pkix-ocsp-nocheck.
certpath: OCSP response is signed by an Authorized Responder
certpath: Verified signature of OCSP Response
certpath: Response's validity interval is from Wed Mar 19 13:57:32 PDT 2014 until Wed Mar 26 13:57:32
PDT 2014
certpath: -checker7 validation succeeded
```

## Further Reading

- [Fixing Certificate Revocation](#)

Next: [Configuring Hostname Verification](#)

# Configuring Hostname Verification

Hostname verification is a little known part of HTTPS that involves a [server identity check](#) to ensure that the client is talking to the correct server and has not been redirected by a man in the middle attack.

The check involves looking at the certificate sent by the server, and verifying that the `dnsName` in the `subjectAltName` field of the certificate matches the host portion of the URL used to make the request.

WS SSL does hostname verification automatically, using the [DefaultHostnameVerifier](#) to implement the [hostname verifier](#) fallback interface.

# Modifying the Hostname Verifier

If you need to specify a different hostname verifier, you can configure `application.conf` to provide your own custom [HostnameVerifier](#):

```
play.ws.ssl.hostnameVerifierClass=org.example.MyHostnameVerifier
```

# Debugging

Hostname Verification can be tested using `dnschef`. A complete guide is out of scope of documentation, but an example can be found in [Testing Hostname Verification](#).

# Further Reading

- [Fixing Hostname Verification](#)

Next: [Example Configurations](#)

# Example Configurations

TLS can be very confusing. Here are some settings that can help.

## Connecting to an internal web service

If you are using WS to communicate with a single internal web service which is configured with an up to date TLS implementation, then you have no need to use an external CA. Internal certificates will work fine, and are arguably [more secure](#) than the CA system. Generate a self signed certificate from the [generating certificates](#) section, and tell the client to trust the CA's public certificate.

```
play.ws.ssl {
 trustManager = {
 stores = [
 { type = "JKS", path = "exampletrust.jks" }
]
 }
}
```

## Connecting to an internal web service with client authentication

If you are using client authentication, then you need to include a keyStore to the key manager that contains a PrivateKeyEntry, which consists of a private key and the X.509

certificate containing the corresponding public key. See the “Configure Client Authentication” section in [generating certificates](#).

```
play.ws.ssl {
 keyManager = {
 stores = [
 { type = "JKS", path = "client.jks", password = "changeit1" }
]
 }
 trustManager = {
 stores = [
 { type = "JKS", path = "exampletrust.jks" }
]
 }
}
```

## Connecting to several external web services

If you are communicating with several external web services, then you may find it more convenient to configure one client with several stores:

```
play.ws.ssl {
 trustManager = {
 stores = [
 { type = "PEM", path = "service1.pem" }
 { path = "service2.jks" }
 { path = "service3.jks" }
]
 }
}
```

If client authentication is required, then you can also set up the key manager with several stores:

```
play.ws.ssl {
 keyManager = {
 stores = [
 { type: "PKCS12", path: "keys/service1-client.p12", password: "changeit1" },
 { type: "PKCS12", path: "keys/service2-client.p12", password: "changeit2" },
 { type: "PKCS12", path: "keys/service3-client.p12", password: "changeit3" }
]
 }
}
```

## Both Private and Public Servers

If you are using WS to access both private and public servers on the same profile, then you will want to include the default JSSE trust store as well:

```
play.ws.ssl {
 trustManager = {
 stores = [
 { path: exampletrust.jks } # Added trust store
 { path: ${java.home}/lib/security/cacerts } # Fallback to default JSSE trust store
]
 }
}
```

Next: [Using the Default SSLContext](#)

# Using the Default SSLContext

If you don't want to use the SSLContext that WS provides for you, and want to use `SSLContext.getDefault`, please set:

```
play.ws.ssl.default = true
```

# Debugging

If you want to debug the default context,

```
play.ws.ssl.debug {
 ssl = true
 sslctx = true
 defaultctx = true
}
```

If you are using the default SSLContext, then the only way to change JSSE behavior is through manipulating the [JSSE system properties](#).

Next: [Debugging SSL](#)

# Debugging SSL Connections

In the event that an HTTPS connection does not go through, debugging JSSE can be a hassle.

WS SSL provides configuration options that will turn on JSSE debug options defined in the [Debugging Utilities](#) and [Troubleshooting Security](#) pages.

To configure, set the `play.ws.ssl.debug` property in `application.conf`:

```
play.ws.ssl.debug = {
 # Turn on all debugging
 all = false
 # Turn on ssl debugging
 ssl = false
 # Turn certpath debugging on
 certpath = false
 # Turn ocsp debugging on
 ocsp = false
 # Enable per-record tracing
 record = false
 # hex dump of record plaintext, requires record to be true
 plaintext = false
 # print raw SSL/TLS packets, requires record to be true
 packet = false
 # Print each handshake message
 handshake = false
 # Print hex dump of each handshake message, requires handshake to be true
 data = false
 # Enable verbose handshake message printing, requires handshake to be true
 verbose = false
 # Print key generation data
 keygen = false
 # Print session activity
 session = false
 # Print default SSL initialization
 defaultctx = false
 # Print SSLContext tracing
 sslctx = false
 # Print session cache tracing
 sessioncache = false
 # Print key manager tracing
 keymanager = false
 # Print trust manager tracing
 trustmanager = false
 # Turn pluggability debugging on
 pluggability = false
}
```

NOTE: This feature changes the setting of the `java.net.debug` system property which is global on the JVM. In addition, this feature [changes static properties at runtime](#), and is only intended for use in development environments.

# Verbose Debugging

To see the behavior of WS, you can configuring the SLF4J logger for debug output:

```
logger.play.api.libs.ws.ssl=DEBUG
```

# Dynamic Debugging

If you are working with WSClient instances created dynamically, you can use the `SSLDebugConfig` class to set up debugging using a builder pattern:

```
val debugConfig = SSLDebugConfig().withKeyManager().withHandshake(data = true, verbose = true)
```

# Further reading

Oracle has a number of sections on debugging JSSE issues:

- [Debugging SSL/TLS connections](#)
- [JSSE Debug Logging With Timestamp](#)
- [How to Analyze Java SSL Errors](#)

Next: [Loose Options](#)

# Loose Options

## We understand

Setting up SSL is not all that fun. It is not lost on anyone that setting up even a single web service with HTTPS involves setting up several certificates, reading boring cryptography documentation and dire warnings.

Despite that, all the security features in SSL are there for good reasons, and turning them off, even for development purposes, has led to even less fun than setting up SSL properly.

## Please read this before turning anything off!

Man in the Middle attacks are well known

The information security community is very well aware of how insecure most internal networks are, and uses that to their advantage. A video discussing attacks detailed a wide range of possible attacks.

## **Man in the Middle attacks are common**

The average company can expect to have seven or eight **Man in the Middle** attacks a year. In some cases, up to 300,000 users can be compromised **over several months**.

## **Attackers have a suite of tools that automatically exploit flaws**

The days of the expert hacker are over. Most security professionals use automated linux environments such as Kali Linux to do penetration testing, packed with hundreds of tools to check for exploits. A video of **Cain & Abel** shows passwords being compromised in less than 20 seconds.

Hackers won't bother to see whether something will "look encrypted" or not. Instead, they'll set up a machine with a toolkit that will run through every possible exploit, and go out for coffee.

## **Security is increasingly important and public**

More and more information flows through computers every day. The public and the media are taking increasing notice of the possibility that their private communications can be intercepted. Google, Facebook, Yahoo, and other leading companies have made secure communication a priority and have devoted millions to ensuring that **data cannot be read**.

## **Ethernet / Password protected WiFi does not provide a meaningful level of security.**

A networking auditing tool such as a **Wifi Pineapple** costs around \$100, picks up all traffic sent over a wifi network, and is so good at intercepting traffic that people have turned it on and started **intercepting traffic accidentally**.

## **Companies have been sued for inadequate security**

PCI compliance is not the only thing that companies have to worry about. The FTC sued **Fandango** and **Credit Karma** on charges that they failed to securely transmit information, including credit card information.

## **Correctly configured HTTPS clients are important**

Sensitive, company confidential information goes over web services. A paper discussing insecurities in WS clients was titled **The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software**, and lists poor default configuration and explicit disabling of security options as the primary reason for exposure. The WS client has been configured as much as possible to be secure by default, and there are example configurations provided for your benefit.

# **Mitigation**

If you must turn on loose options, there are a couple of things you can do to minimize your exposure.

**Custom WSClient:** You can create a custom WSClient specifically for the server, using the `WSConfigParser` together with `ConfigFactory.parseString`, and ensure it is never used outside that context.

**Environment Scoping:** You can define environment variables in HOCON to ensure that any loose options are not hardcoded in configuration files, and therefore cannot escape an development environment.

**Runtime / Deployment Checks:** You can add code to your deployment scripts or program that checks that `play.ws.ssl.loose` options are not enabled in a production environment. The runtime mode can be found in the `Application.mode` method.

# Loose Options

Finally, here are the options themselves.

## Disabling Certificate Verification

**NOTE:** In most cases, people turn off certificate verification because they haven't generated certificates. There are other options besides disabling certificate verification.

- [Quick Start to WS SSL](#) shows how to connect directly to a server using a self signed certificate.
- [Generating X.509 Certificates](#) lists a number of GUI applications that will generate certificates for you.
- [Example Configurations](#) shows complete configuration of TLS using self signed certificates.
- If you want to view your application through HTTPS, you can use [ngrok](#) to proxy your application.
- If you need a certificate authority but don't want to pay money, [StartSSL](#) or [CACert](#) will give you a free certificate.
- If you want a self signed certificate and private key without typing on the command line, you can use [selfsignedcertificate.com](#).

If you've read the above and you still want to completely disable certificate verification, set the following;

```
play.ws.ssl.loose.acceptAnyCertificate=true
```

With certificate verification completely disabled, you are vulnerable to attack from anyone on the network using a tool such as [mitmproxy](#).

## Disabling Weak Ciphers Checking

There are some ciphers which are known to have flaws, and are [disabled](#) in 1.7. WS will throw an exception if a weak cipher is found in the `ws.ssl.enabledCiphers` list. If you specifically want a weak cipher, set this flag:

```
play.ws.ssl.loose.allowWeakCiphers=true
```

With weak cipher checking disabled, you are vulnerable to attackers that use forged certificates, such as [Flame](#).

## Disabling Hostname Verification

If you want to disable hostname verification, you can set a loose flag:

```
play.ws.ssl.loose.disableHostnameVerification=true
```

With hostname verification disabled, a DNS proxy such as `dnschef` can easily intercept communication.

## Disabled Protocols

WS recognizes “SSLv3”, “SSLv2” and “SSLv2Hello” as weak protocols with a number of security issues, and will throw an exception if they are in the `ws.ssl.enabledProtocols` list. Virtually all servers support `TLSv1`, so there is no advantage in using these older protocols.

If you specifically want a weak protocol, set the loose flag to disable the check:

```
play.ws.ssl.loose.allowWeakProtocols=true
```

SSLv2 and SSLv2Hello (there is no v1) are obsolete and usage in the field is down to 25% on the public Internet. SSLv3 is known to have security issues compared to TLS. The only reason to turn this on is if you are connecting to a legacy server, but doing so does not make you vulnerable per se.

[Next: Testing SSL](#)

# Testing SSL

Testing an SSL client not only involves unit and integration testing, but also involves adversarial testing, which tests that an attacker cannot break or subvert a secure connection.

## Unit Testing

Play comes with `play.api.test.WsTestClient`, which provides two methods, `wsCall` and `wsUrl`. It can be helpful to use `PlaySpecification` and `in new WithApplication`

```
"calls index" in new WithApplication() {
 await(wsCall(routes.Application.index()).get())
}
wsUrl("https://example.com").get()
```

## Integration Testing

If you want confirmation that your client is correctly configured, you can call out to [HowsMySSL](#), which has an API to check JSSE settings.

```
import java.io.File

import play.api.{Mode, Environment}
import play.api.libs.json.JsSuccess
import play.api.libs.ws._
import play.api.libs.ws.ning._
import play.api.test._

import com.typesafe.config.{ConfigFactory, ConfigValueFactory}
import scala.concurrent.duration._

class HowsMySSLSpec extends PlaySpecification {

 def createClient(rawConfig: play.api.Configuration): WSClient = {
 val classLoader = Thread.currentThread().getContextClassLoader
 val parser = new WSConfigParser(rawConfig, new Environment(new File("."), classLoader, Mode.Test))
 val clientConfig = new NingWSClientConfig(parser.parse())
 // Debug flags only take effect in JSSE when DebugConfiguration().configure is called.
 //import play.api.libs.ws.ssl.debug.DebugConfiguration
 //clientConfig.ssl.map {
 // _.debug.map(new DebugConfiguration().configure)
 //}
 val builder = new NingAsyncHttpClientConfigBuilder(clientConfig)
 val client = new NingWSClient(builder.build())
 client
 }

 def configToMap(configString: String): Map[String, _] = {
 import scala.collection.JavaConverters._
 ConfigFactory.parseString(configString).root().unwrapped().asScala.toMap
 }

 "WS" should {

 "verify common behavior" in {
 // GeoTrust SSL CA - G2 intermediate certificate not found in cert chain!
 // See https://github.com/jmhodges/howsmysl/issues/38 for details.
 val geoTrustPem =
 """-----BEGIN CERTIFICATE-----\n|MIIETCCA0GgAwIBAgIDAjpjMA0GCSqGSIb3DQEBCUAMEIxCzAJBgNVBAYT\n|AIVTMRYwFAYDVQQKEw1HZW9UcnVzdCBJbmMuMRswGQYDVQQDExJHZW9UcnVz\n|dCBhbG9iYWwgQ0EwHhcNMTIwODI3MjA0MDQwWhcNMjIwNTIwMjA0MDQwWjBE\n|MQswCQYDVQQGEwJVUzEWMBQGA1UEChMNRR2VvVHJ1c3QgSW5jLjEdMBsGA1UE\n|AxMUR2VvVHJ1c3QgU1NMIENBIC0gRzIwggEiMA0GCSqGSIb3DQEBAQUAA4IB\n|DwAwggEKAoIBAQCS5J/IP2Pa3FT+Pzc7WjRxr/X/aVCFOA9jK0HJSFbjJgltY\n|eYT/JHJv8ml/vJbZmnDPqnPUCITDoYZ2+hJ74vm1kfy/XNFCK6PrF62+J58\n|9xD/kkNm7xzU7qFGiBGJSXI6Jc5LavDXHHYaKTzJ5P0ehdzgMWUFRxasCgdL\n|LnBeawanazpsrwUSxLIRJdY+lynwg2xXHNiI78zs/dYS8T/bQLSuDxjTxa9A\n|kl0HXk7+Yhc3iemLdCai7bgK52wVVzWQct3YTSHUQCNCj+6AMRaraFX0DjtU\n|6QRN8MxOgV7pb1JpTr6mFm1C9VH/4AtWPJhPc48Obxoj8cnI2d+87FLXAgMB\n-----END CERTIFICATE-----"""
 }
 }
}
```

```

|AAGjggFUMIIBUDAfBgNVHSMEGDAWgBTaeph0jYn7qwVkBDF9qn1luMrMTjAd
|BgNVHQ4EFgQUEUrQcznVW2kIXLo9v2SaqIscVbwwEgYDVR0TAQH/AgwBgbEB
|/wIBADAObgNVHQ8BAf8EBAMCAQYwOgYDVR0fBDMwMTAv0C2gK4YpaHR0cDov
|L2NyBc5nZW90cnVzdC5jb20vY3Jscy9ndGdsb2JhbC5jcmwwNAYIKwYBBQUH
|AQEEKDAmMCQGCCsGAQUFBzABhhodHRwOi8vb2NzcC5nZW90cnVzdC5jb20w
|TAYDVR0gBEUwQzBBgpghkgBhvFAQc2MDMwMQYIKwYBBQUHAgEWJWh0dHA6
|Ly93d3cuZ2VvdHJ1c3QuY29tL3Jlc291cmNlc9jcHMwKgYDVR0RBCMwIaQf
|MB0xGzAZBgNVBAMTElZlcm1TaWduTVBLSS0yLT11NDANBgkqhkiG9w0BAQUF
|AAOCAQEAPOU9WhuiNyrjRs82lhg8e/GExVeGd0CdNfAS8HgY+yKk3phLeIHm
|TYbjkQ9C47ncoNb/qfixeZeZ0cNsQqWSIOBdDDMYJckrIVPg5akMfUf+f1Ex
|RF73Kh41opQy98nuwLbGmqzemSFqI6A4ZO6jxIhzMjtQzr+t03UepvTp+UJr
|YLLdRf1dVwjOLVDmEjIWE4rylKKbR6iGf9mY5ffldnRk2JG8hBYo2CVEMH6C
|2KyX5MDkFWzbtiQnAioBEoW6MYhYR3TjuNJkpsMyWS4pS0XxW4JLoKaxhgV
|RNAuZAEVaDj59vlmAwxVG52/AECu8EgnTOCAXi25KhV6vGb4NQ==
-----END CERTIFICATE-----
""".stripMargin

val configString = """
//play.ws.ssl.debug=["certpath", "ssl", "trustmanager"]
|play.ws.ssl.protocol="TLSv1"
|play.ws.ssl.enabledProtocols=["TLSv1"]
|
|play.ws.ssl.trustManager = {
| stores = [
| { type: "PEM", data = ${geotrust.pem} }]
| }
|
|"""
"".stripMargin
val rawConfig = ConfigFactory.parseString(configString)
val configWithPem = rawConfig.withValue("geotrust.pem",
ConfigValueFactory.fromAnyRef(geoTrustPem))
val configWithSystemProperties = ConfigFactory.load(configWithPem)
val playConfiguration = play.api.Configuration(configWithSystemProperties)

val client = createClient(playConfiguration)
val response = await(client.url("https://www.howsmyssl.com/a/check").get())(5.seconds)
response.status must be_==(200)

val jsonOutput = response.json
val result = (jsonOutput \ "tls_version").validate[String]
result must beLike {
 case JsSuccess(value, path) =>
 value must_== "TLS 1.0"
}
}
}
}
}


```

Note that if you are writing tests that involve custom configuration such as revocation checking or disabled algorithms, you may need to pass system properties into SBT:

```
javaOptions in Test += Seq("-Dcom.sun.security.enableCRLDP=true", "-Dcom.sun.net.ssl.checkRevocation=true", "-Djavax.net.debug=all")
```

# Adversarial Testing

There are several points of where a connection can be attacked. Writing these tests is fairly easy, and running these adversarial tests against unsuspecting programmers can be extremely satisfying.

**NOTE:** This should not be taken as a complete list, but as a guide. In situations where security is paramount, a review should be done by professional info-sec consultants.

## Testing Certificate Verification

Write a test to connect to "<https://example.com>". The server should present a certificate which says the subjectAltName is dnsName, but the certificate should be signed by a CA certificate which is not in the trust store. The client should reject it.

This is a very common failure. There are a number of proxies like [mitmproxy](#) or [Fiddler](#) which will only work if certificate verification is disabled or the proxy's certificate is explicitly added to the trust store.

## Testing Weak Cipher Suites

The server should send a cipher suite that includes NULL or ANON cipher suites in the handshake. If the client accepts it, it is sending unencrypted data.

**NOTE:** For a more in depth test of a server's cipher suites, see [sslyze](#).

## Testing Certificate Validation

To test for weak signatures, the server should send the client a certificate which has been signed with, for example, the MD2 digest algorithm. The client should reject it as being too weak.

To test for weak certificate, The server should send the client a certificate which contains a public key with a key size under 1024 bits. The client should reject it as being too weak.

**NOTE:** For a more in depth test of certification validation, see [tlspretense](#) and [frankencert](#).

## Testing Hostname Verification

Write a test to "<https://example.com>". If the server presents a certificate where the subjectAltName's dnsName is not example.com, the connection should terminate.

**NOTE:** For a more in depth test, see [dnschef](#).

**Next:** [Databases](#)

# H2 database

The H2 in memory database is very convenient for development because your evolutions are run from scratch when play is restarted. If you are using anorm you probably need it to closely mimic your planned production database. To tell h2 that you want to mimic a particular database you add a parameter to the database url in your application.conf file, for example:

```
db.default.url="jdbc:h2:mem:play;MODE=MYSQL"
```

## Target databases

MySQL      MODE=MYSQL

H2 does not have a `uuid()` function. You can use `random_uuid()` instead. Or insert the following line into your `1.sql` file:

```
CREATE ALIAS UUID FOR
"org.h2.value.ValueUuid.getNewRandom";
Text comparison in MySQL is case insensitive by default, while
in H2 it is case sensitive (as in most other databases). H2 does
support case insensitive text comparison, but it needs to be set
separately, using SET IGNORECASE TRUE. This affects
comparison using =, LIKE, REGEXP.
```

DB2      MODE=DB2

Derby      MODE=DERBY

HSQldb      MODE=HSQldb

MS SQL      MODE=MSSQLServer

Oracle      MODE=Oracle

PostgreSQL      MODE=PostgreSQL

## Prevent in memory DB reset

H2, by default, drops your in memory database if there are no connections to it anymore. You probably don't want this to happen. To prevent this add `DB_CLOSE_DELAY=-1` to the url (use a semicolon as a separator)

eg:`jdbc:h2:mem:play;MODE=MYSQL;DB_CLOSE_DELAY=-1`

## Caveats

H2, by default, creates tables with upper case names. Sometimes you don't want this to happen, for example when using H2 with Play evolutions in some compatibility modes. To prevent this add `DATABASE_TO_UPPER=FALSE` to the url (use a semicolon as a separator) eg:  
eg:  
`jdbc:h2:mem:play;MODE=PostgreSQL;DB_CLOSE_DELAY=-1;DATABASE_TO_UPPER=FALSE`

---

## H2 Browser

You can browse the contents of your database by typing `h2-browser` at the play console. An SQL browser will run in your web browser.

---

## H2 Documentation

More H2 documentation is available [from their web site](#).

Next: [Managing database evolutions](#)

# Managing database evolutions

When you use a relational database, you need a way to track and organize your database schema evolutions. Typically there are several situations where you need a more sophisticated way to track your database schema changes:

- When you work within a team of developers, each person needs to know about any schema change.
  - When you deploy on a production server, you need to have a robust way to upgrade your database schema.
  - If you work on several machines, you need to keep all database schemas synchronized.
- 

## Enable evolutions

Add `evolutions` into your dependencies list. For example, in `build.sbt`:

```
libraryDependencies += evolutions
```

## Evolutions scripts

Play tracks your database evolutions using several evolutions script. These scripts are written in plain old SQL and should be located in the `conf/evolutions/{database}`

`name}` directory of your application. If the evolutions apply to your default database, this path is `conf/evolutions/default`.

The first script is named `1.sql`, the second script `2.sql`, and so on...

Each script contains two parts:

- The **Ups** part that describes the required transformations.
- The **Downs** part that describes how to revert them.

For example, take a look at this first evolution script that bootstrap a basic application:

```
Users schema

--- !Ups

CREATE TABLE User (
 id bigint(20) NOT NULL AUTO_INCREMENT,
 email varchar(255) NOT NULL,
 password varchar(255) NOT NULL,
 fullname varchar(255) NOT NULL,
 isAdmin boolean NOT NULL,
 PRIMARY KEY (id)
);

--- !Downs

DROP TABLE User;
```

As you see you have to delimit the both Ups and Downs section by using comments in your SQL script.

Play splits your `.sql` files into a series of semicolon-delimited statements before executing them one-by-one against the database. So if you need to use a semicolon *within* a statement, escape it by entering `;;` instead of `;`. For example, `INSERT INTO punctuation(name, character) VALUES ('semicolon', ';;');`.

Evolutions are automatically activated if a database is configured in `application.conf` and evolution scripts are present. You can disable them by setting `play.evolutions.enabled=false`. For example when tests set up their own database you can disable evolutions for the test environment.

When evolutions are activated, Play will check your database schema state before each request in DEV mode, or before starting the application in PROD mode. In DEV mode, if your database schema is not up to date, an error page will suggest that you synchronise your database schema by running the appropriate SQL script.

The screenshot shows a web browser window with the URL `localhost:9000`. The title bar says "Database 'default' needs evolution!". Below the title, a message states "An SQL script will be run on your database - **Apply this script now!**". A dark grey box contains the text "This SQL script must be run:" followed by a numbered SQL script:

```
1 # --- Rev:1,Ups - 9984b4b
2 CREATE TABLE User (
3 id bigint(20) NOT NULL AUTO_INCREMENT,
4 email varchar(255) NOT NULL,
5 password varchar(255) NOT NULL,
6 fullname varchar(255) NOT NULL,
7 isAdmin boolean NOT NULL,
8 PRIMARY KEY (id)
9);
```

If you agree with the SQL script, you can apply it directly by clicking on the 'Apply evolutions' button.

## Evolutions configuration

Evolutions can be configured both globally and per datasource. For global configuration, keys should be prefixed with `play.evolutions`. For per datasource configuration, keys should be prefixed with `play.evolutions.db.<datasourcename>`, for example `play.evolutions.db.default`. The following configuration options are supported:

- `enabled` - Whether evolutions are enabled. If configured globally to be false, it disables the evolutions module altogether. Defaults to true.

- `autocommit` - Whether autocommit should be used. If false, evolutions will be applied in a single transaction. Defaults to true.
- `useLocks` - Whether a locks table should be used. This must be used if you have many Play nodes that may potentially run evolutions, but you want to ensure that only one does. It will create a table called `play evolutions lock`, and use a `SELECT FOR UPDATE NOWAIT` or `SELECT FOR UPDATE` to lock it. This will only work for Postgres, Oracle, and MySQL InnoDB. It will not work for other databases. Defaults to false.
- `autoApply` - Whether evolutions should be automatically applied. In dev mode, this will cause both ups and downs evolutions to be automatically applied. In prod mode, it will cause only ups evolutions to be automatically applied. Defaults to false.
- `autoApplyDowns` - Whether down evolutions should be automatically applied. In prod mode, this will cause down evolutions to be automatically applied. Has no effect in dev mode. Defaults to false. For example, to enable `autoApply` for all evolutions, you might `set play.evolutions.autoApply=true` in `application.conf` or in a system property. To disable autocommit for a datasource named `default`, you `set play.evolutions.db.default.autocommit=false`.

# Synchronizing concurrent changes

Now let's imagine that we have two developers working on this project. Developer A will work on a feature that requires a new database table. So he will create the following `2.sql` evolution script:

```
Add Post

--- !Ups
CREATE TABLE Post (
 id bigint(20) NOT NULL AUTO_INCREMENT,
 title varchar(255) NOT NULL,
 content text NOT NULL,
 postedAt date NOT NULL,
 author_id bigint(20) NOT NULL,
 FOREIGN KEY (author_id) REFERENCES User(id),
 PRIMARY KEY (id)
);

--- !Downs
DROP TABLE Post;
```

Play will apply this evolution script to Developer A's database.

On the other hand, developer B will work on a feature that requires altering the User table. So he will also create the following `2.sql` evolution script:

```
Update User
```

```
--- !Ups
ALTER TABLE User ADD age INT;

--- !Downs
ALTER TABLE User DROP age;
```

Developer B finishes his feature and commits (let's say they are using Git). Now developer A has to merge the his colleague's work before continuing, so he runs git pull, and the merge has a conflict, like:

```
Auto-merging db/evolutions/2.sql
CONFLICT (add/add): Merge conflict in db/evolutions/2.sql
Automatic merge failed; fix conflicts and then commit the result.
```

Each developer has created a `2.sql` evolution script. So developer A needs to merge the contents of this file:

```
<<<<< HEAD
Add Post
```

```
--- !Ups
CREATE TABLE Post (
 id bigint(20) NOT NULL AUTO_INCREMENT,
 title varchar(255) NOT NULL,
 content text NOT NULL,
 postedAt date NOT NULL,
 author_id bigint(20) NOT NULL,
 FOREIGN KEY (author_id) REFERENCES User(id),
 PRIMARY KEY (id)
);
```

```
--- !Downs
DROP TABLE Post;
=====
```

```
Update User
```

```
--- !Ups
ALTER TABLE User ADD age INT;

--- !Downs
ALTER TABLE User DROP age;
>>>>> devB
```

The merge is really easy to do:

```
Add Post and update User

--- !Ups
ALTER TABLE User ADD age INT;

CREATE TABLE Post (
 id bigint(20) NOT NULL AUTO_INCREMENT,
 title varchar(255) NOT NULL,
```

```

content text NOT NULL,
postedAt date NOT NULL,
author_id bigint(20) NOT NULL,
FOREIGN KEY (author_id) REFERENCES User(id),
PRIMARY KEY (id)
);

--- !Downs
ALTER TABLE User DROP age;

DROP TABLE Post;

```

This evolution script represents the new revision 2 of the database, that is different of the previous revision 2 that developer A has already applied.

So Play will detect it and ask developer A to synchronize his database by first reverting the old revision 2 already applied, and by applying the new revision 2 script:

## Inconsistent states

Sometimes you will make a mistake in your evolution scripts, and they will fail. In this case, Play will mark your database schema as being in an inconsistent state and will ask you to manually resolve the problem before continuing.

For example, the Ups script of this evolution has an error:

```

Add another column to User

--- !Ups
ALTER TABLE Userxxx ADD company varchar(255);

--- !Downs
ALTER TABLE User DROP company;

```

So trying to apply this evolution will fail, and Play will mark your database schema as inconsistent:

The screenshot shows a web browser window with the URL `localhost:9000`. The title bar says "Database 'default' is in inconsistent state!". The main content area has a red background with white text: "Database 'default' is in inconsistent state!". Below that, in a pink box, it says "An evolution has not been applied properly. Please check the problem and resolve it before making it as resolved - [Mark it resolved](#)". Underneath, in a black box, it says "We got the following error: Table 'test.userxxx' doesn't exist [ERROR:1146, SQLSTATE 42S02] while trying to run this SQL script:". A code block shows the SQL script:

```
1 # --- Rev:1,Ups - 7361418
2
3 ALTER TABLE Userxxx ADD company varchar(255);
```

Now before continuing you have to fix this inconsistency. So you run the fixed SQL command:

```
ALTER TABLE User ADD company varchar(255);
```

... and then mark this problem as manually resolved by clicking on the button.

But because your evolution script has errors, you probably want to fix it. So you modify the `3.sql` script:

```
Add another column to User

--- !Ups
ALTER TABLE User ADD company varchar(255);

--- !Downs
ALTER TABLE User DROP company;
```

Play detects this new evolution that replaces the previous 3 one, and will run the appropriate script. Now everything is fixed, and you can continue to work.

In development mode however it is often simpler to simply trash your development database and reapply all evolutions from the beginning.

## Transactional DDL

By default, each statement of each evolution script will be executed immediately. If your database supports [Transactional DDL](#) you can set `evolutions.autocommit=false` in application.conf to change this behaviour, causing **all** statements to be executed in **one transaction** only. Now, when an evolution script fails to apply with autocommit disabled, the whole transaction gets rolled back and no changes will be applied at all. So your database stays “clean” and will not become inconsistent. This allows you to easily fix any DDL issues in the evolution scripts without having to modify the database by hand like described above.

## Evolution storage and limitations

Evolutions are stored in your database in a table called `play_evolutions`. A Text column stores the actual evolution script. Your database probably has a 64kb size limit on a text column. To work around the 64kb limitation you could: manually alter the `play_evolutions` table structure changing the column type or (preferred) create multiple evolutions scripts less than 64kb in size.

Next: [Deploying your application](#)

# Deploying your application

We have seen how to run a Play application in development mode, however the `run` command should not be used to run an application in production mode. When using `run`, on each request, Play checks with sbt to see if any files have changed, and this may have significant performance impacts on your application.

There are several ways to deploy a Play application in production mode. Let's start by using the recommended way, creating a distribution artifact.

## The application secret

Before you run your application in production mode, you need to generate an application secret. To read more about how to do this, see [Configuring the application secret](#). In the examples below, you will see the use of `-Dapplication.secret=abcdefghijklm`. You must generate your own secret to use here.

# Using the dist task

The dist task builds a binary version of your application that you can deploy to a server without any dependency on sbt or activator, the only thing the server needs is a Java installation.

In the Play console, simply type `dist`:

```
[my-first-app] $ dist
Terminal
$ activator
[info] Loading global plugins from /Users/jroper/.sbt/0.13/plugins
[info] Loading project definition from /Users/jroper/tmp/my-first-app/project
[info] Set current project to my-first-app (in build file:/Users/jroper/tmp/my-first-app/)
[my-first-app] $ dist
[info] Updating {file:/Users/jroper/tmp/my-first-app/}my-first-app...
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT-sources.jar ...
[info] Done packaging.
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Wrote /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT.pom
[info] Compiling 5 Scala sources and 1 Java source to /Users/jroper/tmp/my-first-app/target/scala-2.10/classes...
[info] Main Scala API documentation to /Users/jroper/tmp/my-first-app/target/scala-2.10/api...
model contains 17 documentable templates
[info] Main Scala API documentation successful.
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT.jar
...
[info] Done packaging.
[info]
[info] Your package is ready in /Users/jroper/tmp/my-first-app/target/universal/my-first-app-1.0-SNAPSHOT.zip
[info]
[success] Total time: 13 s, completed 28/03/2014 3:26:01 PM
[my-first-app] $
```

This produces a ZIP file containing all JAR files needed to run your application in the `target/universal` folder of your application.

To run the application, unzip the file on the target server, and then run the script in the `bin` directory. The name of the script is your application name, and it comes in two versions, a bash shell script, and a windows `.bat` script.

```
$ unzip my-first-app-1.0.zip
$ my-first-app-1.0/bin/my-first-app -Dplay.crypto.secret=abcdefghijklm
```

You can also specify a different configuration file for a production environment, from the command line:

```
$ my-first-app-1.0/bin/my-first-app -Dconfig.file=/full/path/to/conf/application-prod.conf
```

For a full description of usage invoke the start script with a `-h` option.

For Unix users, zip files do not retain Unix file permissions so when the file is expanded the start script will be required to be set as an executable:

```
$ chmod +x /path/to/bin/<project-name>
```

Alternatively a tar.gz file can be produced instead. Tar files retain permissions. Invoke the `universal:packageZipTarball` task instead of the `dist` task:

```
activator universal:packageZipTarball
```

By default, the `dist` task will include the API documentation in the generated package. If this is not necessary, add these lines in `build.sbt`:

```
sources in (Compile, doc) := Seq.empty
```

```
publishArtifact in (Compile, packageDoc) := false
```

For builds with sub-projects, the statement above has to be applied to all sub-project definitions.

## The Native Packager

Play uses the [SBT Native Packager plugin](#). The native packager plugin declares the `dist` task to create a zip file. Invoking the `dist` task is directly equivalent to invoking the following:

```
[my-first-app] $ universal:packageBin
```

Many other types of archive can be generated including:

- tar.gz
- OS X disk images
- Microsoft Installer (MSI)
- RPMs
- Debian packages
- System V / init.d and Upstart services in RPM/Debian packages

Please consult the [documentation](#) on the native packager plugin for more information.

### Build a server distribution

The sbt-native-packager plugins provides a number archetypes. The one that Play uses by default is called the Java server archetype, which enables the following features:

- System V or Upstart startup scripts
- [Default folders](#)

A full documentation can be found in the [documentation](#).

*Minimal Debian settings*

Add the following settings to your build:

```

lazy val root = (project in file("."))
 .enablePlugins(PlayScala, DebianPlugin)

maintainer in Linux := "First Lastname <first.last@example.com>"

packageSummary in Linux := "My custom package summary"

packageDescription := "My longer package description"

```

Then build your package with:

```
[my-first-app] $ debian:packageBin
```

*Minimal RPM settings*

Add the following settings to your build:

```

lazy val root = (project in file("."))
 .enablePlugins(PlayScala, RpmPlugin)

maintainer in Linux := "First Lastname <first.last@example.com>"

packageSummary in Linux := "My custom package summary"

packageDescription := "My longer package description"

rpmRelease := "1"

rpmVendor := "example.com"

rpmUrl := Some("http://github.com/example/server")

rpmLicense := Some("Apache v2")

```

Then build your package with:

```
[my-first-app] $ rpm:packageBin
```

There will be some error logging. This is because rpm logs to stderr instead of stdout.

## Including additional files in your distribution

Anything included in your project's `dist` directory will be included in the distribution built by the native packager. Note that in Play, the `dist` directory is equivalent to the `src/universal` directory mentioned in the native packager's own documentation.

# Play PID Configuration

Play manages its own PID, which is described in the [Production configuration](#). In order to tell the startup script where to place the PID file put a file `application.ini` inside the `dist/conf` folder and add the following content:

```
-Dpidfile.path=/var/run/${{app_name}}/play.pid
Add all other startup settings here, too
```

For a full list of replacements take a closer look at the [customize java server documentation](#) and [customize java app documentation](#).

## Publishing to a Maven (or Ivy) repository

You can also publish your application to a Maven repository. This publishes both the JAR file containing your application and the corresponding POM file.

You have to configure the repository you want to publish to, in your `build.sbt` file:

```
publishTo := Some(
 "My resolver" at "https://mycompany.com/repo"
)

credentials += Credentials(
 "Repo", "https://mycompany.com/repo", "admin", "admin123"
)
```

Then in the Play console, use the `publish` task:

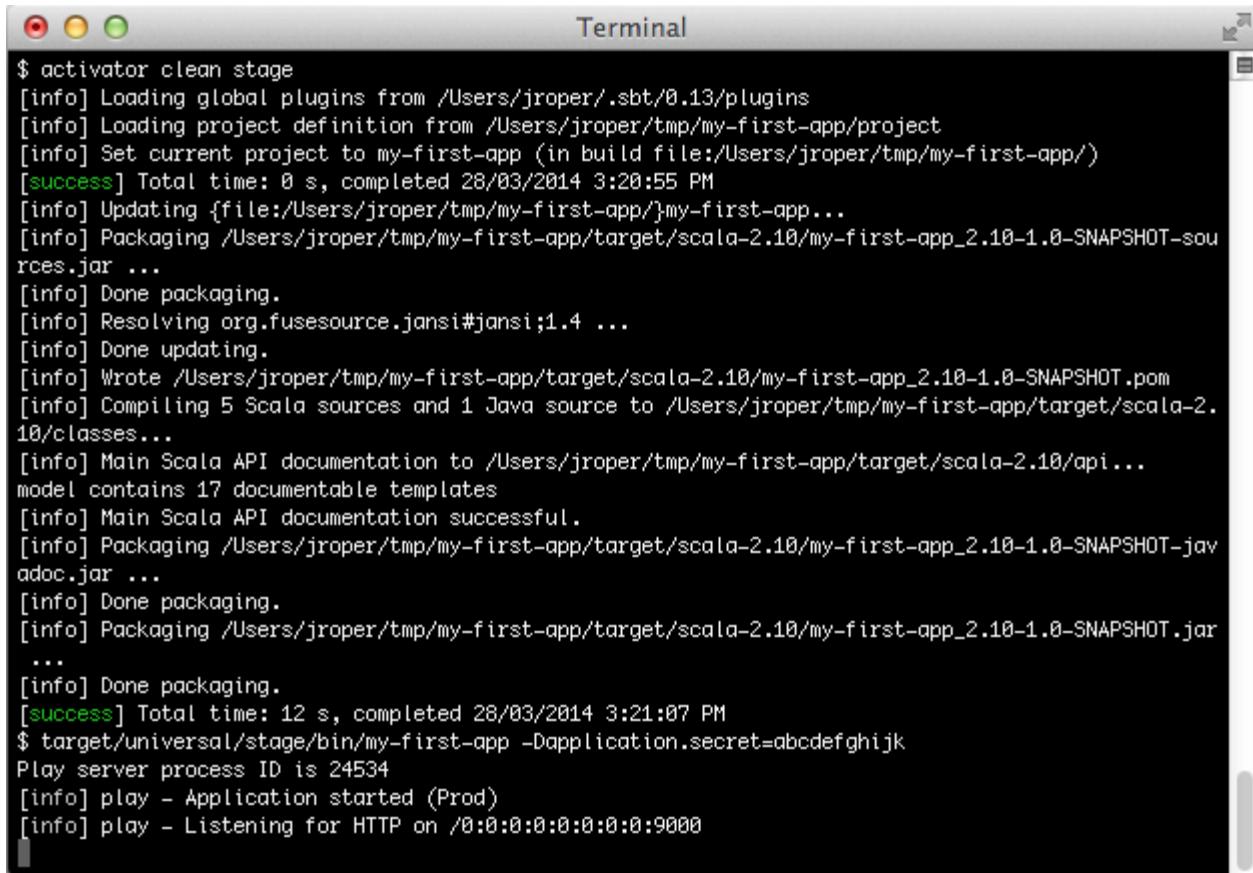
```
[my-first-app] $ publish
```

Check the [sbt documentation](#) to get more information about the resolvers and credentials definition.

## Running a production server in place

In some circumstances, you may not want to create a full distribution, you may in fact want to run your application from your project's source directory. This requires an sbt or activator installation on the server, and can be done using the `stage` task.

```
$ activator clean stage
```



```
$ activator clean stage
[info] Loading global plugins from /Users/jroper/.sbt/0.13/plugins
[info] Loading project definition from /Users/jroper/tmp/my-first-app/project
[info] Set current project to my-first-app (in build file:/Users/jroper/tmp/my-first-app/)
[success] Total time: 0 s, completed 28/03/2014 3:20:55 PM
[info] Updating {file:/Users/jroper/tmp/my-first-app/}my-first-app...
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT-sources.jar ...
[info] Done packaging.
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Wrote /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT.pom
[info] Compiling 5 Scala sources and 1 Java source to /Users/jroper/tmp/my-first-app/target/scala-2.10/classes...
[info] Main Scala API documentation to /Users/jroper/tmp/my-first-app/target/scala-2.10/api...
model contains 17 documentable templates
[info] Main Scala API documentation successful.
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[info] Packaging /Users/jroper/tmp/my-first-app/target/scala-2.10/my-first-app_2.10-1.0-SNAPSHOT.jar ...
[info] Done packaging.
[success] Total time: 12 s, completed 28/03/2014 3:21:07 PM
$ target/universal/stage/bin/my-first-app -Dapplication.secret=abcdefghijkl
Play server process ID is 24534
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

This cleans and compiles your application, retrieves the required dependencies and copies them to the `target/universal/stage` directory. It also creates a `bin/<start>` script where `<start>` is the project's name. The script runs the Play server on Unix style systems and there is also a corresponding `bat` file for Windows.

For example to start an application of the project `my-first-app` from the project folder you can:

```
$ target/universal/stage/bin/my-first-app -Dplay.crypto.secret=abcdefghijkl
```

You can also specify a different configuration file for a production environment, from the command line:

```
$ target/universal/stage/bin/my-first-app -Dconfig.file=/full/path/to/conf/application-prod.conf
```

## Running a test instance

Play provides a convenient utility for running a test application in prod mode.

This is not intended for production usage.

To run an application in prod mode, run `testProd`:

```
[my-first-app] $ testProd
```

# Using the SBT assembly plugin

Though not officially supported, the SBT assembly plugin may be used to package and run Play applications. This will produce one jar as an output artifact, and allow you to execute it directly using the `java` command.

To use this, add a dependency on the plugin to your `project/plugins.sbt` file:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

Now add the following configuration to your `build.sbt`:

```
import AssemblyKeys._
```

```
assemblySettings
```

```
mainClass in assembly := Some("play.core.server.NettyServer")
```

```
fullClasspath in assembly += Attributed.blank(PlayKeys.playPackageAssets.value)
```

Now you can build the artifact by running `activator assembly`, and run your application by running:

```
$ java -jar target/scala-2.XX/<yourprojectname>-assembly-<version>.jar -Dplay.crypto.secret=abcdefghijklm
```

You'll need to substitute in the right project name, version and scala version, of course.

**Next:** [Production configuration](#)

# Additional configuration

There are a number of different types of configuration that you can configure in production.

The three main types are:

- General configuration
- Logging configuration
- JVM configuration

Each of these types have different methods to configure them.

## General configuration

Play has a number of configurable settings. You can configure database connection URLs, the application secret, the HTTP port, SSL configuration, and so on.

Most of Play's configuration is defined in various `.conf` files, which use the HOCON format. The main configuration file that you'll use is the `application.conf` file. You can find this file at `conf/application.conf` within your project. The `application.conf` file is loaded from the classpath at runtime (or you can override where it is loaded from). There can only be one `application.conf` per project.

Other `.conf` files are loaded too. Libraries define default settings in `reference.conf` files. These files are stored in the libraries' JARs—one `reference.conf` per JAR—and aggregated together at runtime. The `reference.conf` files provide defaults; they are overridden by any settings defined in the `application.conf` file.

Play's configuration can also be defined using system properties and environment variables. This can be handy when settings change between environments; you can use the `application.conf` for common settings, but use system properties and environment variables to change settings when you run the application in different environments.

System properties override settings in `application.conf`, and `application.conf` overrides the default settings in the various `reference.conf` files.

You can override runtime configuration in several ways. This can be handy when settings vary between environments; you can change the configuration dynamically for each environment. Here are your choices for runtime configuration:

- Using an alternate `application.conf` file.
- Overriding individual settings using system properties.
- Injecting configuration values using environment variables.

## Specifying an alternate configuration file

The default is to load the `application.conf` file from the classpath. You can specify an alternative configuration file if needed:

*Using `-Dconfig.resource`*

This will search for an alternative configuration file in the application classpath (you usually provide these alternative configuration files into your application `conf/` directory before packaging). Play will look into `conf/` so you don't have to add `conf/`.

```
$ /path/to/bin/<project-name> -Dconfig.resource=prod.conf
```

*Using `-Dconfig.file`*

You can also specify another local configuration file not packaged into the application artifacts:

```
$ /path/to/bin/<project-name> -Dconfig.file=/opt/conf/prod.conf
```

Note that you can always reference the original configuration file in a new `prod.conf` file using the `include` directive, such as:

```
include "application.conf"
```

```
key.to.override=blah
```

## Overriding configuration with system properties

Sometimes you don't want to specify another complete configuration file, but just override a bunch of specific keys. You can do that by specifying them as Java System properties:

```
$ /path/to/bin/<project-name> -Dplay.crypto.secret=abcdefghijkl -Ddb.default.password=toto
```

### *Specifying the HTTP server address and port using system properties*

You can provide both HTTP port and address easily using system properties. The default is to listen on port `9000` at the `0.0.0.0` address (all addresses).

```
$ /path/to/bin/<project-name> -Dhttp.port=1234 -Dhttp.address=127.0.0.1
```

### *Changing the path of RUNNING\_PID*

It is possible to change the path to the file that contains the process id of the started application. Normally this file is placed in the root directory of your play project, however it is advised that you put it somewhere where it will be automatically cleared on restart, such as `/var/run`:

```
$ /path/to/bin/<project-name> -Dpidfile.path=/var/run/play.pid
```

Make sure that the directory exists and that the user that runs the Play application has write permission for it.

Using this file, you can stop your application using the `kill` command, for example:

```
$ kill $(cat /var/run/play.pid)
```

## Using environment variables

You can also reference environment variables from your `application.conf` file:

```
my.key = defaultvalue
```

```
my.key = ${?MY_KEY_ENV}
```

Here, the override field `my.key = ${?MY_KEY_ENV}` simply vanishes if there's no value for `MY_KEY_ENV`, but if you set an environment variable `MY_KEY_ENV` for example, it would be used.

## Server configuration options

A full list of server configuration options, including defaults, can be seen here:

```
play {
 server {

 # The root directory for the Play server instance. This value can
 # be set by providing a path as the first argument to the Play server
 # launcher script. See `ServerConfig.loadConfiguration`.
 dir = ${?user.dir}

 # HTTP configuration
 http {
 # The HTTP port of the server. Use a value of "disabled" if the server
```

```

shouldn't bind an HTTP port.
port = 9000
port = ${?http.port}

The interface address to bind to.
address = "0.0.0.0"
address = ${?http.address}
}

HTTPS configuration
https {

The HTTPS port of the server.
port = ${?https.port}

The interface address to bind to
address = "0.0.0.0"
address = ${?https.address}

The SSL engine provider
engineProvider = "play.core.server.ssl.DefaultSSLEngineProvider"
engineProvider = ${?play.http.sslengineprovider}

HTTPS keystore configuration, used by the default SSL engine provider
keyStore {
 # The path to the keystore
 path = ${?https.keyStore}

 # The type of the keystore
 type = "JKS"
 type = ${?https.keyStoreType}

 # The password for the keystore
 password = ""
 password = ${?https.keyStorePassword}

 # The algorithm to use. If not set, uses the platform default algorithm.
 algorithm = ${?https.keyStoreAlgorithm}
}

HTTPS truststore configuration
trustStore {

 # If true, does not do CA verification on client side certificates
 noCaVerification = false
}

The type of ServerProvider that should be used to create the server.
If not provided, the ServerStart class that instantiates the server
will provide a default value.
provider = ${?server.provider}

```

```

The path to the process id file created by the server when it runs.
If set to "/dev/null" then no pid file will be created.
pidfile.path = ${play.server.dir}/RUNNING_PID
pidfile.path = ${?pidfile.path}

Configuration options specific to Netty
netty {
 # The maximum length of the initial line. This effectively restricts the maximum length of a URL that the
server will
 # accept, the initial line consists of the method (3-7 characters), the URL, and the HTTP version (8
characters),
 # including typical whitespace, the maximum URL length will be this number - 18.
 maxInitialLineLength = 4096
 maxInitialLineLength = ${?http.netty.maxInitialLineLength}

 # The maximum length of the HTTP headers. The most common effect of this is a restriction in cookie
length, including
 # number of cookies and size of cookie values.
 maxHeaderSize = 8192
 maxHeaderSize = ${?http.netty.maxHeaderSize}

 # The maximum length of body bytes that Netty will read into memory at a time.
 # This is used in many ways. Note that this setting has no relation to HTTP chunked transfer encoding -
Netty will
 # read "chunks", that is, byte buffers worth of content at a time and pass it to Play, regardless of whether
the body
 # is using HTTP chunked transfer encoding. A single HTTP chunk could span multiple Netty chunks if it
exceeds this.
 # A body that is not HTTP chunked will span multiple Netty chunks if it exceeds this or if no content
length is
 # specified. This only controls the maximum length of the Netty chunk byte buffers.
 maxChunkSize = 8192
 maxChunkSize = ${?http.netty.maxChunkSize}

 # Whether the Netty wire should be logged
 log.wire = false
 log.wire = ${?http.netty.log.wire}

 # Netty options. Possible keys here are defined by:
 #
 # http://netty.io/3.9/api/org/jboss/netty/channel/socket/SocketChannelConfig.html
 # http://netty.io/3.9/api/org/jboss/netty/channel/socket/ServerSocketChannelConfig.html
 # http://netty.io/3.9/api/org/jboss/netty/channel/socket/nio/NioSocketChannelConfig.html
 #
 # Options that pertain to the listening server socket are defined at the top level, options for the sockets
associated
 # with received client connections are prefixed with child.*
 option {

 # Set whether connections should use TCP keep alive
 # child.keepAlive = false

 # Set whether the TCP no delay flag is set
 }
}

```

```

child.tcpNoDelay = false

Set the size of the backlog of TCP connections. The default and exact meaning of this parameter is JDK
specific.
 # backlog = 100
}
}
}

Configuration specific to Play's experimental Akka HTTP backend
akka {
 # How long to wait when binding to the listening socket
 http-bind-timeout = 5 seconds
}
}

```

# Logging configuration

Logging can be configured by creating a logback configuration file. This can be used by your application through the following means:

## Bundling a custom logback configuration file with your application

Create an alternative logback config file called `logback.xml` and copy that to `<app>/conf`

You can also specify another logback configuration file via a System property. Please note that if the configuration file is not specified then play will use the default `logback.xml` that comes with play in the production mode. This means that any log level settings in `application.conf` file will be overridden. As a good practice always specify your `logback.xml`.

## Using `-Dlogger.resource`

Specify another logback configuration file to be loaded from the classpath:

```
$ /path/to/bin/<project-name> -Dlogger.resource=conf/prod-logger.xml
```

## Using `-Dlogger.file`

Specify another logback configuration file to be loaded from the file system:

```
$ /path/to/bin/<project-name> -Dlogger.file=/opt/prod/prod-logger.xml
```

## Using `-Dlogger.url`

Specify another logback configuration file to be loaded from an URL:

```
$ /path/to/bin/<project-name> -Dlogger.url=http://conf.mycompany.com/logger.xml
```

# JVM configuration

You can specify any JVM arguments to the application startup script. Otherwise the default JVM settings will be used:

```
$ /path/to/bin/<project-name> -J-Xms128M -J-Xmx512m -J-server
```

As a convenience you can also set memory min, max, permgen and the reserved code cache size in one go; a formula is used to determine these values given the supplied parameter (which represents maximum memory):

```
$ /path/to/bin/<project-name> -mem 512 -J-server
```

**Next:** Setting up a front end HTTP server

# Setting up a front end HTTP server

You can easily deploy your application as a stand-alone server by setting the application HTTP port to 80:

```
$ /path/to/bin/<project-name> -Dhttp.port=80
```

Note that you probably need root permissions to bind a process on this port.

However, if you plan to host several applications in the same server or load balance several instances of your application for scalability or fault tolerance, you can use a front end HTTP server.

Note that using a front end HTTP server will rarely give you better performance than using Play server directly. However, HTTP servers are very good at handling HTTPS, conditional GET requests and static assets, and many services assume a front end HTTP server is part of your architecture.

## Set up with lighttpd

This example shows you how to configure [lighttpd](#) as a front end web server. Note that you can do the same with Apache, but if you only need virtual hosting or load balancing, lighttpd is a very good choice and much easier to configure!

The `/etc/lighttpd/lighttpd.conf` file should define things like this:

```
server.modules = (
 "mod_access",
 "mod_proxy",
 "mod_accesslog"
)
...
$HTTP["host"] =~ "www.myapp.com" {
 proxy.balance = "round-robin" proxy.server = ("/" =>
 (("host" => "127.0.0.1", "port" => 9000))
)
}
$HTTP["host"] =~ "www.loadbalancedapp.com" {
 proxy.balance = "round-robin" proxy.server = ("/" =>
 ("host" => "127.0.0.1", "port" => 9001),
 ("host" => "127.0.0.1", "port" => 9002)
)
}
```

## Set up with nginx

This example shows you how to configure [nginx](#) as a front end web server. Note that you can do the same with Apache, but if you only need virtual hosting or load balancing, nginx is a very good choice and much easier to configure!

The `/etc/nginx/nginx.conf` file should define things like this:

```
worker_processes 1;

events {
 worker_connections 1024;
}

http {
 include mime.types;
 default_type application/octet-stream;

 sendfile on;
 keepalive_timeout 65;

 proxy_buffering off;
 proxy_set_header X-Real-IP $remote_addr;
 proxy_set_header X-Forwarded-Proto $scheme;
 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
 proxy_set_header Host $http_host;
 proxy_http_version 1.1;

 upstream my-backend {
 server 127.0.0.1:9000;
```

```

}

server {
 listen 80;
 server_name www.mysite.com;
 location / {
 proxy_pass http://my-backend;
 }
}

#server {
listen 443;
ssl on;
#
http://www.selfsignedcertificate.com/ is useful for development testing
ssl_certificate /etc/ssl/certs/my_ssl.crt;
ssl_certificate_key /etc/ssl/private/my_ssl.key;
#
From https://bettercrypto.org/static/applied-crypto-hardening.pdf
ssl_prefer_server_ciphers on;
ssl_protocols TLSv1 TLSv1.1 TLSv1.2; # not possible to do exclusive
ssl_ciphers
'EDH+CAMELLIA:EDH+aRSA:EECDH+aRSA+AESGCM:EECDH+aRSA+SHA384:EECDH+aRSA+SHA
256:EECDH:+CAMELLIA256:+AES256:+CAMELLIA128:+AES128:+SSLv3:!aNULL:!eNULL:!LOW:!3D
ES:!MD5:!EXP:!PSK:!DSS:!RC4:!SEED:!ECDSA:CAMELLIA256-SHA:AES256-SHA:CAMELLIA128-
SHA:AES128-SHA';
add_header Strict-Transport-Security max-age=15768000; # six months
use this only if all subdomains support HTTPS!
add_header Strict-Transport-Security "max-age=15768000; includeSubDomains"
#
keepalive_timeout 70;
server_name www.mysite.com;
location / {
proxy_pass http://my-backend;
}
#}
}

Note Make sure you are using version 1.2 or greater of Nginx otherwise chunked responses won't
work properly.

```

## Set up with Apache

The example below shows a simple set up with [Apache httpd server](#) running in front of a standard Play configuration.

`LoadModule proxy_module modules/mod_proxy.so`

```

...
<VirtualHost *:80>
 ProxyPreserveHost On
 ServerName www.loadbalancedapp.com
 ProxyPass /excluded !
 ProxyPass / http://127.0.0.1:9000/

```

```
ProxyPassReverse / http://127.0.0.1:9000/
</VirtualHost>
```

## Advanced proxy settings

When using an HTTP frontal server, request addresses are seen as coming from the HTTP server. In a usual set-up, where you both have the Play app and the proxy running on the same machine, the Play app will see the requests coming from 127.0.0.1.

Proxy servers can add a specific header to the request to tell the proxied application where the request came from. Most web servers will add an X-Forwarded-For header with the remote client IP address as first argument. If the proxy server is running on localhost and connecting from 127.0.0.1, Play will trust its `X-Forwarded-For` header.

However, the host header is untouched, it'll remain issued by the proxy. If you use Apache 2.x, you can add a directive like:

```
ProxyPreserveHost on
```

The host: header will be the original host request header issued by the client. By combining these two techniques, your app will appear to be directly exposed.

If you don't want this play app to occupy the whole root, add an exclusion directive to the proxy config:

```
ProxyPass /excluded !
```

## Apache as a front proxy to allow transparent upgrade of your application

The basic idea is to run two Play instances of your web application and let the front-end proxy load-balance them. In case one is not available, it will forward all the requests to the available one.

Let's start the same Play application two times: one on port 9999 and one on port 9998.

```
$ start -Dhttp.port=9998
$ start -Dhttp.port=9999
```

Now, let's configure our Apache web server to have a load balancer.

In Apache, I have the following configuration:

```
<VirtualHost mysuperwebapp.com:80>
ServerName mysuperwebapp.com
<Location /balancer-manager>
SetHandler balancer-manager
Order Deny,Allow
Deny from all
Allow from .mysuperwebapp.com
</Location>
<Proxy balancer://mycluster>
BalancerMember http://localhost:9999
BalancerMember http://localhost:9998 status=+H
</Proxy>
<Proxy *>
Order Allow,Deny
Allow From All
</Proxy>
ProxyPreserveHost On
ProxyPass /balancer-manager !
ProxyPass / balancer://mycluster/
ProxyPassReverse / balancer://mycluster/
</VirtualHost>
```

The important part is `balancer://mycluster`. This declares a load balancer. The `+H` option means that the second Play application is on standby. But you can also instruct it to load balance.

Apache also provides a way to view the status of your cluster. Simply point your browser to `/balancer-manager` to view the current status of your clusters.

Because Play is completely stateless you don't have to manage sessions between the 2 clusters. You can actually easily scale to more than 2 Play instances.

Note that [Apache does not support Websockets](#), so you may wish to use another front end proxy (such as [HAProxy](#) or Nginx) that does implement this functionality.

Note that [ProxyPassReverse might rewrite incorrectly headers](#) adding an extra `/` to the URIs, so you may wish to use this workaround:

```
ProxyPassReverse / http://localhost:9999 ProxyPassReverse /
http://localhost:9998
```

# Configure trusted proxies

To determine the client IP address Play has to know which are the trusted proxies in your network.

Those can be configured with `play.http.forwarded.trustedProxies`. You can define a list of proxies and/or subnet masks that Play recognizes as belonging to your network. Default is `127.0.0.1` and `::FF`

There exists two possibilities how proxies are set in the HTTP-headers:

- the legacy method with X-Forwarded headers
- the RFC 7239 with Forwarded headers

The type of header to parse is set via `play.http.forwarded.version`. Valid values are `x-forwarded` or `rfc7239`.

The default is `x-forwarded`.

For more information, please read the [RFC 7239](#).

Next: [Configuring HTTPS](#)

# Configuring HTTPS

Play can be configured to serve HTTPS. To enable this, simply tell Play which port to listen to using the `https.port` system property. For example:

```
./start -Dhttps.port=9443
```

## Providing configuration

HTTPS configuration can either be supplied using system properties or in `application.conf`. For more details see the [configuration](#) and [production configuration](#) pages.

## SSL Certificates

### SSL Certificates from a keystore

By default, Play will generate itself a self-signed certificate, however typically this will not be suitable for serving a website. Play uses Java key stores to configure SSL certificates and keys.

Signing authorities often provide instructions on how to create a Java keystore (typically with reference to Tomcat configuration). The official Oracle documentation on how to generate keystores using the JDK keytool utility can be found [here](#). There is also an example in the [Generating X.509 Certificates](#) section.

Having created your keystore, the following configuration properties can be used to configure Play to use it:

- **play.server.https.keyStore.path** - The path to the keystore containing the private key and certificate, if not provided generates a keystore for you
- **play.server.https.keyStore.type** - The key store type, defaults to `JKS`
- **play.server.https.keyStore.password** - The password, defaults to a blank password
- **play.server.https.keyStore.algorithm** - The key store algorithm, defaults to the platforms default algorithm

## SSL Certificates from a custom SSL Engine

Another alternative to configure the SSL certificates is to provide a custom `SSLEngine`. This is also useful in cases where a customized SSLEngine is required, such as in the case of client authentication.

*in Java, an implementation must be provided*

*for `play.server.SSLEngineProvider`*

```
import play.server.ApplicationProvider;
import play.server.SSLEngineProvider;

import javax.net.ssl.*;
import java.security.NoSuchAlgorithmException;

public class CustomSSLEngineProvider implements SSLEngineProvider {
 private ApplicationProvider applicationProvider;

 public CustomSSLEngineProvider(ApplicationProvider applicationProvider) {
 this.applicationProvider = applicationProvider;
 }

 @Override
 public SSLEngine createSSLEngine() {
 try {
 // change it to your custom implementation
 return SSLContext.getDefault().createSSLEngine();
 } catch (NoSuchAlgorithmException e) {
 throw new RuntimeException(e);
 }
 }
}
```

*in Scala, an implementation must be provided*

*for `play.server.api.SSLEngineProvider`*

```
import javax.net.ssl._
import play.core.ApplicationProvider
import play.server.api._

class CustomSSLEngineProvider(appProvider: ApplicationProvider) extends SSLEngineProvider {

 override def createSSLEngine(): SSLEngine = {
 // change it to your custom implementation
 SSLContext.getDefault.createSSLEngine()
 }
}
```

```
}
```

```
}
```

Having created an implementation

for `play.server.SSLEngineProvider` or `play.server.api.SSLEngineProvider`,

the following system property configures Play to use it:

- `play.server.https.engineProvider` - The path to the class implementing `play.server.SSLEngineProvider` or `play.server.api.SSLEngineProvider`:

Example:

```
./start -Dhttps.port=9443 -Dplay.server.https.engineProvider=mypackage.CustomSSLEngineProvider
```

## Turning HTTP off

To disable binding on the HTTP port, set the `http.port` system property to be `disabled`, eg:

```
./start -Dhttp.port=disabled -Dhttps.port=9443 -Dplay.server.https.keyStore.path=/path/to/keystore -
Dplay.server.https.keyStore.password=changeme
```

## Production usage of HTTPS

If Play is serving HTTPS in production, it should be running JDK 1.8. JDK 1.8 provides a number of new features that make JSSE feasible as a [TLS termination layer](#). If not using JDK 1.8, using a [reverse proxy](#) in front of Play will give better control and security of HTTPS.

If you intend to use Play for TLS termination layer, please note the following settings:

- `SSLParameters.setUseCipherSuiteOrder()` - Reorders cipher suite order to the server's preference.
- `-Djdk.tls.ephemeralDHKeySize=2048` - Increases the key size in a DH key exchange.
- `-Djdk.tls.rejectClientInitiatedRenegotiation=true` - Rejects client renegotiation.

Next: [Deploying to a cloud service](#)

## Deploying a Play application to a cloud service

Many third party cloud services have built in support for deploying Play applications.

- Deploying to Heroku
- Deploying to Cloud Foundry
- Deploying to Clever Cloud
- Deploying to Boxfuse and AWS

Next: Deploying to Heroku

# Deploying to Heroku

Heroku is a cloud application platform – a way of building and deploying web apps.

To get started:

1. Install the Heroku Toolbelt
2. Sign up for a Heroku account

There are two methods of deployment to Heroku:

- Pushing to a remote Git repository.
- Using the sbt-heroku plugin.

## Deploying to a remote Git repository

Store your application in git

```
$ git init
$ git add .
$ git commit -m "init"
```

Create a new application on Heroku

```
$ heroku create
Creating warm-frost-1289... done, stack is cedar-14
http://warm-frost-1289.herokuapp.com/ | git@heroku.com:warm-frost-1289.git
Git remote heroku added
```

This provisions a new application with an HTTP (and HTTPS) endpoint and Git endpoint for your application. The Git endpoint is set as a new remote named `heroku` in your Git repository's configuration.

Deploy your application

To deploy your application on Heroku, use Git to push it into the `heroku` remote repository:

```
$ git push heroku master
Counting objects: 93, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (84/84), done.
Writing objects: 100% (93/93), 1017.92 KiB | 0 bytes/s, done.
```

```

Total 93 (delta 38), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: ----> Play 2.x - Scala app detected
remote: ----> Installing OpenJDK 1.8... done
remote: ----> Priming Ivy cache (Scala-2.11, Play-2.4)... done
remote: ----> Running: sbt compile stage
...
remote: ----> Dropping ivy cache from the slug
remote: ----> Dropping sbt boot dir from the slug
remote: ----> Dropping compilation artifacts from the slug
remote: ----> Discovering process types
remote: Procfile declares types -> web
remote:
remote: ----> Compressing... done, 93.3MB
remote: ----> Launching... done, v6
remote: https://warm-frost-1289.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/warm-frost-1289.git
 * [new branch] master -> master
Heroku will run sbt stage to prepare your application. On the first deployment, all
dependencies will be downloaded, which takes a while to complete (but they will be cached
for future deployments).

```

If you are using RequireJS and you find that your application hangs at this step:

[info] Optimizing JavaScript with RequireJS

Then try following the steps in the [Using Node.js to Perform JavaScript Optimization for Play and Scala Applications](#) on the Heroku Dev Center. This will greatly improve the performance of the Javascript engine.

## Check that your application has been deployed

Now, let's check the state of the application's processes:

```
$ heroku ps
==== web (Free): `target/universal/stage/bin/sample-app -Dhttp.port=${PORT}`
web.1: up 2015/01/09 11:27:51 (~ 4m ago)
```

The web process is up and running. We can view the logs to get more information:

```
$ heroku logs
2015-07-13T20:44:47.358320+00:00 heroku[web.1]: Starting process with command
`target/universal/stage/bin/myapp -Dhttp.port=${PORT}`
2015-07-13T20:44:49.750860+00:00 app[web.1]: Picked up JAVA_TOOL_OPTIONS: -Xmx384m -Xss512k
-Dfile.encoding=UTF-8
2015-07-13T20:44:52.297033+00:00 app[web.1]: [warn] application - Logger configuration in conf files is
deprecated and has no effect. Use a logback configuration file instead.
```

```
2015-07-13T20:44:54.960105+00:00 app[web.1]: [info] p.a.l.c.ActorSystemProvider - Starting application
default Akka system: application
2015-07-13T20:44:55.066582+00:00 app[web.1]: [info] play.api.Play$ - Application started (Prod)
2015-07-13T20:44:55.445021+00:00 heroku[web.1]: State changed from starting to up
2015-07-13T20:44:55.330940+00:00 app[web.1]: [info] p.c.s.NettyServer$ - Listening for HTTP on
/0:0:0:0:0:0:0:8626
...
...
```

We can also tail the logs as we would for a regular file. This is useful for debugging:

```
$ heroku logs -t --app warm-frost-1289
2015-07-13T20:44:47.358320+00:00 heroku[web.1]: Starting process with command
`target/universal/stage/bin/myapp -Dhttp.port=${PORT}`
2015-07-13T20:44:49.750860+00:00 app[web.1]: Picked up JAVA_TOOL_OPTIONS: -Xmx384m -Xss512k
-Dfile.encoding=UTF-8
2015-07-13T20:44:52.297033+00:00 app[web.1]: [warn] application - Logger configuration in conf files is
deprecated and has no effect. Use a logback configuration file instead.
2015-07-13T20:44:54.960105+00:00 app[web.1]: [info] p.a.l.c.ActorSystemProvider - Starting application
default Akka system: application
2015-07-13T20:44:55.066582+00:00 app[web.1]: [info] play.api.Play$ - Application started (Prod)
2015-07-13T20:44:55.445021+00:00 heroku[web.1]: State changed from starting to up
2015-07-13T20:44:55.330940+00:00 app[web.1]: [info] p.c.s.NettyServer$ - Listening for HTTP on
/0:0:0:0:0:0:0:8626
...
...
```

Looks good. We can now visit the app by running:

```
$ heroku open
```

# Deploying with the sbt-heroku plugin

The Heroku sbt plugin utilizes an API to provide direct deployment of prepackaged standalone web applications to Heroku. This may be a preferred approach for applications that take a long time to compile, or that need to be deployed from a Continuous Integration server such as Travis CI or Jenkins.

## Adding the plugin

To include the plugin in your project, add the following to your `project/plugins.sbt` file:

```
addSbtPlugin("com.heroku" % "sbt-heroku" % "0.5.1")
```

Next, we must configure the name of the Heroku application the plugin will deploy to. But first, create a new app. Install the Heroku Toolbelt and run the `create` command.

```
$ heroku create
Creating obscure-sierra-7788... done, stack is cedar-14
```

```
http://obscure-sierra-7788.herokuapp.com/ | git@heroku.com:obscure-sierra-7788.git
```

Now add something like this to your `build.sbt`, but replace “obscure-sierra-7788” with the name of the application you created (or you can skip this if you are using Git locally). `herokuAppName` in `Compile := "obscure-sierra-7788"`

The sbt-heroku project’s documentation contains details on [configuring the execution of the plugin](#).

## Deploying with the plugin

With the plugin added, you can deploy to Heroku by running this command:

```
$ sbt stage deployHeroku
...
[info] -----> Packaging application...
[info] - app: obscure-sierra-7788
[info] - including: target/universal/stage/
[info] -----> Creating build...
[info] - file: target/heroku/slug.tgz
[info] - size: 30MB
[info] -----> Uploading slug... (100%)
[info] - success
[info] -----> Deploying...
[info] remote:
[info] remote: -----> Fetching custom tar buildpack... done
[info] remote: -----> sbt-heroku app detected
[info] remote: -----> Installing OpenJDK 1.8... done
[info] remote: -----> Discovering process types
[info] remote: Procfile declares types -> console, web
[info] remote:
[info] remote: -----> Compressing... done, 78.9MB
[info] remote: -----> Launching... done, v6
[info] remote: https://obscure-sierra-7788.herokuapp.com/ deployed to Heroku
[info] remote:
[info] -----> Done
[success] Total time: 90 s, completed Aug 29, 2014 3:36:43 PM
```

And you can visit your application by running this command:

```
$ heroku open -a obscure-sierra-7788
```

You can see the logs for your application by running this command:

```
$ heroku logs -a obscure-sierra-7788
```

Note that if you are using Git, you can omit the `-a` option above as the app name will be detected from the Git remote that was added to your config when you ran `heroku create`.

# Connecting to a database

Heroku provides a number of relational and NoSQL databases through [Heroku Add-ons](#). Play applications on Heroku are automatically provisioned with a [Heroku Postgres](#) database. To configure your Play application to use the Heroku Postgres database, first add the PostgreSQL JDBC driver to your application dependencies (`build.sbt`):

```
libraryDependencies += "org.postgresql" % "postgresql" % "9.4-1201-jdbc41"
```

Then create a new file in your project's root directory named `Procfile` (with a capital "P") that contains the following (substituting the `myapp` with your project's name):

```
web: target/universal/stage/bin/myapp -Dhttp.port=${PORT} -Dplay.evolutions.db.default.autoApply=true -Ddb.default.url=${DATABASE_URL}
```

This instructs Heroku that for the process named `web` it will run Play and override the `play.evolutions.db.default.autoApply`, `db.default.driver`, and `db.default.url` configuration parameters. Note that the `Procfile` command can be maximum 255 characters long. Alternatively, use the `-Dconfig.resource=` or `-Dconfig.file=` mentioned in [production configuration](#) page.

Also, be aware the the `DATABASE_URL` is in the platform independent format:

```
vendor://username:password@host:port/db
```

Play will automatically convert this into a JDBC URL for you if you are using one of the built in database connection pools. But other database libraries and frameworks, such as Slick or Hibernate, may not support this format natively. If that's the case, you may try using the experimental `JDBC_DATABASE_URL` in place of `DATABASE_URL` in the configuration like this:

```
db.default.url=${?JDBC_DATABASE_URL}
db.default.username=${?JDBC_DATABASE_USERNAME}
db.default.password=${?JDBC_DATABASE_PASSWORD}
```

Note that the creation of a `Procfile` is not actually required by Heroku, as Heroku will look in your Play application's `conf` directory for an `application.conf` file in order to determine that it is a Play application.

## Further learning resources

- [Getting Started with Scala and Play on Heroku](#)
- [Deploying Scala and Play Applications with the Heroku sbt Plugin](#)
- [Using Node.js to Perform JavaScript Optimization for Play and Scala Applications](#)
- [Deploy Scala and Play Applications to Heroku from Travis CI](#)
- [Deploy Scala and Play Applications to Heroku from Jenkins CI](#)
- [Running a Remote sbt Console for a Scala or Play Application](#)
- [Using WebSockets on Heroku with Java and the Play Framework](#)
- [Seed Project for Play and Heroku](#)
- [Play Tutorial for Java](#)
- [Getting Started with Play, Scala, and Squeryl](#)
- [Edge Caching With Play, Heroku, and CloudFront](#)
- [Optimizing Play for Database-Driven Apps](#)
- [Play App with a Scheduled Job on Heroku](#)
- [Using Amazon S3 for File Uploads with Java and Play](#)

**Next:** [Deploying to Cloud Foundry](#)

# Deploying to CloudFoundry / AppFog

## Prerequisites

Sign up for a free [Cloud Foundry](#) account and install or update the Cloud Foundry command line tool, VMC, to the latest version (0.3.18 or higher) by using the following command:  
gem install vmc

## Build your Application

Package your app by typing the `dist` command in the play prompt.

## Deploy your Application

Deploy the created zip file to Cloud Foundry with the VMC push command. If you choose to create a database service, Cloud Foundry will automatically apply your database evolutions on application start.

```
yourapp$ vmc push --path=dist/yourapp-1.0.zip
Application Name: yourapp
Detected a Play Framework Application, is this correct? [Yn]:
Application Deployed URL [yourapp.cloudfoundry.com]:
Memory reservation (128M, 256M, 512M, 1G, 2G) [256M]:
How many instances? [1]:
Create services to bind to 'yourapp'? [yN]: y
1: mongodb
2: mysql
3: postgresql
4: rabbitmq
5: redis
What kind of service?: 3
Specify the name of the service [postgresql-38199]: your-db
Create another? [yN]:
Would you like to save this configuration? [yN]: y
Manifest written to manifest.yml.
Creating Application: OK
Creating Service [your-db]: OK
Binding Service [your-db]: OK
Uploading Application:
Checking for available resources: OK
```

```
Processing resources: OK
Packing application: OK
Uploading (186K): OK
Push Status: OK
Staging Application 'yourapp': OK
Starting Application 'yourapp': OK
```

# Working With Services

## Auto-Reconfiguration

Cloud Foundry uses a mechanism called auto-reconfiguration to automatically connect your Play application to a relational database service. If a single database configuration is found in the Play configuration (for example, `default`) and a single database service instance is bound to the application, Cloud Foundry will automatically override the connection properties in the configuration to point to the PostgreSQL or MySQL service bound to the application.

This is a great way to get simple apps up and running quickly. However, it is quite possible that your application will contain SQL that is specific to the type of database you are using. In these cases, or if your app needs to bind to multiple services, you may choose to avoid auto-reconfiguration and explicitly specify the service connection properties.

## Connecting to Cloud Foundry Services

As always, Cloud Foundry provides all of your service connection information to your application in JSON format through the `VCAP_SERVICES` environment variable. However, connection information is also available as series of properties you can use in your Play configuration. Here is an example of connecting to a PostgreSQL service named `tasks-db` from within an `application.conf` file:

```
db.default.driver=${?cloud.services.tasks-db.connection.driver}
db.default.url=${?cloud.services.tasks-db.connection.url}
db.default.password=${?cloud.services.tasks-db.connection.password}
db.default.username=${?cloud.services.tasks-db.connection.username}
```

This information is available for all types of services, including NoSQL and messaging services. Also, if there is only a single service of a type (e.g. `postgresql`), you can refer to that service only by type instead of specifically by name, as exemplified below:

```
db.default.driver=${?cloud.services.postgresql.connection.driver}
db.default.url=${?cloud.services.postgresql.connection.url}
db.default.password=${?cloud.services.postgresql.connection.password}
db.default.username=${?cloud.services.postgresql.connection.username}
```

We recommend keeping these properties in a separate file (for example `cloud.conf`) and then including them only when building a distribution for Cloud Foundry. You can specify an alternative config file to `play dist` by using `-Dconfig.file`.

## Opting out of Auto-Reconfiguration

If you use the properties referenced above, you will automatically be opted-out. To explicitly opt out, include a file named “cloudfoundry.properties” in your application’s conf directory, and add the entry `autoconfig=false`

Next: [Deploying to Clever Cloud](#)

# Deploying to Clever Cloud

Clever Cloud is a Platform as a Service solution. You can deploy on it Scala, Java, PHP, Python and Node.js applications. Its main particularity is that it supports **automatic vertical and horizontal scaling**.

Clever Cloud supports Play! 2 applications natively. The present guide explains how to deploy your application on Clever Cloud.

## Create a new application on Clever Cloud

Create your Play! application on Clever Cloud [dashboard](#).

## Deploy your application

To deploy your application on Clever Cloud, just use git to push your code to the application remote repository.

```
$ git remote add <your-remote-name> <your-git-deployment-url>
$ git push <your-remote-name> master
```

**Important tip: do not forget to push to the remote master branch.**

If you work in a different branch, just use:

```
$ git remote add <your-remote-name> <your-git-deployment-url>
$ git push <your-remote-name> <your-branch-name>:master
```

Clever Cloud will run `sbt update stage` to prepare your application. On the first

deployment, all dependencies will be downloaded, which takes a while to complete (but will be cached for future deployments).

---

# Check the deployment of your application

You can check the deployment of your application by visiting the *logs* section of your application in the dashboard.

---

## [Optional] Configure your application

You can custom your application with a `clevercloud/play.json` file.

The file must contain the following fields:

```
{
 "deploy": {
 "goal": <string>
 }
}
```

That field can contain additional configuration like:

```
"-Dconfig.resource=clevercloud.conf", "-Dplay.version=2.0.4" or "-
Dplay.evolutions.autoApply=true".
```

---

## Connecting to a database

Just go to the *Services* section in the Clever Cloud dashboard to add the database you need: MySQL, PostgreSQL or Couchbase.

As in every Play! 2 application, the only file you have to modify is your `conf/application.conf` file.

### Example: setup MySQL database

```
db.default.url="jdbc:mysql://{yourcleverdbhost}/{dbname}"
db.default.driver=com.mysql.jdbc.Driver
db.default.username={yourcleveruser}
db.default.password={yourcleverpass}
```

## Further information

If you need further information, just check our complete [documentation](#).

Next: [Deploying to Boxfuse and AWS](#)

# Deploying to Boxfuse and AWS

Boxfuse lets you deploy your Play applications on AWS. It is based on 3 core principles: Immutable Infrastructure, Minimal Images and Blue/Green deployments.

Boxfuse comes with native Play application support and works by turning your Play dist zip into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. This image is generated on-the-fly in seconds and is about 100x smaller than a regular Linux system. It literally only contains your Play application, a JRE and the Linux kernel, reducing the security attack surface to the minimum possible.

Boxfuse works with your AWS account and automatically provisions all the necessary AWS resources your application requires including AMIs, Elastic IPs, Elastic Load Balancers, Security Groups, Auto-Scaling Groups and EC2 instances.

## Prerequisites

Sign up for a free [Boxfuse](#) account as well as a free [AWS](#) account and [install the Boxfuse command line client](#).

As Boxfuse works with your AWS account, it first needs the necessary permissions to do so. So if you haven't already done so, go to the Boxfuse Console and [connect your AWS account](#) now.

## Build your Application

Package your app using the Typesafe Activator by typing the `activator dist` command in your project directory.

## Deploy your Application

Every new Boxfuse account comes with 3 environments: `dev`, `test` and `prod`. `dev` is for fast roundtrips locally on VirtualBox environment and `test` and `prod` are on AWS.

So let's deploy the new zip file of your application to the `prod` environment on AWS:

```
myapp$ boxfuse run -env=prod

Fusing Image for myapp-1.0.zip ...
Image fused in 00:09.817s (75949 K) -> myuser/myapp:1.0
Pushing myuser/myapp:1.0 ...
Verifying myuser/myapp:1.0 ...
Waiting for AWS to create an AMI for myuser/myapp:1.0 in eu-central-1 (this may take up to 50 seconds) ...
AMI created in 00:34.152s in eu-central-1 -> ami-8b988be7
Creating security group boxsg-myuser-prod-myapp-1.0 ...
Launching t2.micro instance of myuser/myapp:1.0 (ami-8b988be7) in prod (eu-central-1) ...
Instance launched in 00:35.372s -> i-ebea4857
Waiting for AWS to boot Instance i-ebea4857 and Payload to start at http://52.29.129.239/ ...
Payload started in 00:50.316s -> http://52.29.129.239/
Remapping Elastic IP 52.28.107.167 to i-ebea4857 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
Deployment completed successfully. myuser/myapp:1.0 is up and running at http://myapp-myuser.boxfuse.io/
```

You can now visit your app deployed on AWS by running:

```
myapp$ boxfuse open -env=prod
```

## Further learning resources

- [Get Started with Boxfuse & Play](#)
- [Deploy Play Framework Scala Apps effortlessly to AWS](#)
- [Boxfuse Play integration reference documentation](#)

Next: [Experimental libraries](#)

# Akka HTTP server backend(*experimental*)

Play experimental libraries are not ready for production use. APIs may change. Features may not work properly.

Play 2's main server is built on top of [Netty](#). In Play 2.4 we started experimenting with an experimental server based on [Akka HTTP](#). Akka HTTP is an HTTP library built on top of Akka. It is written by the authors of [Spray](#).

The purpose of this backend is:

- to check that the Akka HTTP API provides all the features that Play needs
- to gain knowledge about Akka HTTP in case we want to use it in Play in the future.

In future versions of Play we may implement a production quality Akka HTTP backend, but in Play 2.4 the Akka HTTP server is mostly a proof of concept. We do **not** recommend that you use it for anything other than learning about Play or Akka HTTP server code. In Play 2.4 you should always use the default Netty-based server for production code.

# Known issues

- Slow. There is a lot more copying in the Akka HTTP backend because the Play and Akka HTTP APIs are not naturally compatible. A lot of extra copying is needed to translate the objects.
- WebSockets are not supported, due to missing support in Akka HTTP.
- No HTTPS support, again due to missing support in Akka HTTP.
- Server shutdown is a bit rough. HTTP server actors are just killed.
- The implementation contains code duplicated from the Netty backend.

# Usage

To use the Akka HTTP server backend you first need to disable the Netty server and add the Akka HTTP server plugin to your project:

```
lazy val root = (project in file("."))
 .enablePlugins(PlayScala, PlayAkkaHttpServer)
 .disablePlugins(PlayNettyServer)
```

Now Play should automatically select the Akka HTTP server for running in dev mode, prod and in tests.

## Manually selecting the Akka HTTP server

If for some reason you have both the Akka HTTP server and the Netty HTTP server on your classpath, you'll need to manually select it. This can be done using the `play.server.provider` system property, for example, in dev mode:

```
run -Dplay.server.provider=play.core.server.akkahttp.AkkaHttpServerProvider
```

## Verifying that the Akka HTTP server is running

When the Akka HTTP server is running it will tag all requests with a tag called `HTTP_SERVER` with a value of `akka-http`. The Netty backend will not have a value for this tag.

```
Action { request =>
 assert(request.tags.get("HTTP_SERVER") == Some("akka-http"))
 ...
}
```

## Configuring the Akka HTTP server

The Akka HTTP server is configured with Typesafe Config, like the rest of Play. This is the default configuration for the Akka HTTP backend. The `log-dead-letters` setting is set to `off` because the Akka HTTP server can send a lot of letters. If you want this on then you'll need to enable it in your `application.conf`.

```
play {

 # The server provider class name
 server.provider = "play.core.server.akkahttp.AkkaHttpServerProvider"
```

```

akka {
 # How long to wait when binding to the listening socket
 http-bind-timeout = 5 seconds
}

}

akka {

 # Turn off dead letters until Akka HTTP server is stable
 log-dead-letters = off

}

```

**Note:** In dev mode, when you use the `run` command, your `application.conf` settings will not be picked up by the server. This is because in dev mode the server starts before the application classpath is available. There are several [other options](#) you'll need to use instead.

Next: [Reactive Streams integration](#)

# Reactive Streams integration (experimental)

Play experimental libraries are not ready for production use. APIs may change. Features may not work properly.

[Reactive Streams](#) is a new standard that gives a common API for asynchronous streams. Play 2.4 introduces some wrappers to convert Play's [Iteratees and Enumerators](#) into Reactive Streams objects. This means that Play can integrate with other software that supports Reactive Streams, e.g. [Akka Streams](#), [RxJava](#) and [others](#).

The purpose of the API is:

- to check that the Reactive Streams API is powerful enough to express Play iteratees and enumerators
- to test integration between Play and Akka Streams
- to provide stream conversions needed by the experimental [Akka HTTP server backend](#)
- to test out an API.

This API is **highly experimental**. It should be reasonably free of bugs, but its methods and classes and concepts are very likely to change in the future.

## Known issues

- No Java API. This shouldn't be hard to implement, but it hasn't been done yet.

- The implementation hasn't been tested against the Reactive Streams test suite so there may be some conformance issues.
  - May need to lift `Input` events into the stream to ensure that `Input.EOF` events cannot be lost and to provide proper support for `Input.Empty`. At the moment there is the potential for event loss when adapting iteratees and enumerators.
  - No performance tuning has been done.
  - Needs support for two-way conversion between all the main stream and iteratee types.
  - Documentation is limited.
- 

# Usage

Include the Reactive Streams integration library into your project.

```
libraryDependencies += "com.typesafe.play" %% "play-streams-experimental" % "2.4.x"
```

All access to the module is through the `Streams` object.

Here is an example that adapts a `Future` into a single-element `Publisher`.

```
val fut: Future[Int] = Future { ... }
val pubr: Publisher[Int] = Streams.futureToPublisher(fut)
```

See the `Streams` object's [API documentation](#) for more information.

For more examples you can look at the code used by the experimental [Akka HTTP server backend](#). Here are the main files where you can find examples:

- [ModelConversion](#)
- [AkkaStreamsConversion](#)
- [AkkaHttpServer](#)

Next: [Hacking Play](#)

# Building Play from source

To benefit from the latest improvements and bug fixes after the initial beta release, you may want to compile Play from source. You'll need a [Git client](#) to fetch the source.

---

## Grab the source

From the shell, first checkout the Play source:

```
$ git clone git://github.com/playframework/playframework.git
```

Then go to the `playframework/framework` directory and launch the `build` script to enter the sbt build console:

```
$ cd playframework/framework
```

```
$./build
```

```
> publishLocal
```

This will build and publish Play for the default Scala version (currently 2.10.5). If you want to publish for all versions, you can cross build:

```
> +publishLocal
```

Or to publish for a specific Scala version:

```
> +++2.11.6 publishLocal
```

## Build the documentation

Documentation is available at `playframework/documentation` as Markdown files. To see HTML, run the following:

```
$ cd playframework/documentation
$./build run
```

To see documentation at <http://localhost:9000/@documentation>

For more details on developing the Play documentation, see the [Documentation Guidelines](#).

## Run tests

You can run basic tests from the sbt console using the `test` task:

```
> test
```

Like with publishing, you can prefix the command with `+` to run the tests against all supported Scala versions.

The Play PR validation runs a few more tests than just the basic tests, including scripted tests, testing the documentation code samples, and testing the Play activator templates. To run all the tests, run the `framework/runtests` script:

```
$ cd playframework/framework
$./runtests
```

## Use in projects

Compiling and running projects using the Play version you have built from source requires some custom configuration.

Navigate to your existing Play project and make the following edits in `project/plugins.sbt`:

```
// Change the sbt plugin to use the local Play build (2.4.0-SNAPSHOT)
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.0-SNAPSHOT")
```

Once you have done this, you can start the console and interact with your project normally:

```
$ cd <projectdir>
$ activator
```

# Using Code in Eclipse

You can find at [Stackoverflow](#) some information how to setup eclipse to work on the code.

Next: [Repositories](#)

# Artifact repositories

## Typesafe repository

All Play artifacts are published to the Typesafe repository  
at <https://repo.typesafe.com/typesafe/releases/>.

**Note:** it's a Maven2 compatible repository.

To enable it in your sbt build, you must add a proper resolver (typically in `plugins.sbt`):

```
// The Typesafe repository
resolvers += "Typesafe Releases" at "https://repo.typesafe.com/typesafe/releases/"
```

## Accessing snapshots

Snapshots are published daily from our [Continuous Integration Server](#) to the Typesafe snapshots repository at <https://repo.typesafe.com/typesafe/snapshots/>.

**Note:** it's an ivy style repository.

```
// The Typesafe snapshots repository
```

```
resolvers += Resolver.url("Typesafe Ivy Snapshots Repository", url("https://repo.typesafe.com/typesafe/ivy-snapshots"))(Resolver.ivyStylePatterns)
```

Next: [Issue tracker](#)

# Issues tracker

We use GitHub as our issue tracker, at:

<https://github.com/playframework/playframework/issues>.

# Reporting bugs

Bug reports are incredibly helpful, so take time to report bugs and request features in our ticket tracker. We're always grateful for patches to Play's code. Indeed, bug reports with attached patches will get fixed far quickly than those without any.

Please include as much relevant information as possible including the exact framework version you're using and a code snippet that reproduces the problem.

Don't have too many expectations. Unless the bug is really a serious 'everything is broken' thing, you're creating a ticket to start a discussion. Having a patch (or a branch on Github we can pull from) is better, but then again we'll only pull high-quality branches that make sense to be in the core of Play.

**Next:** Documentation guidelines

# Guidelines for writing Play documentation

The Play documentation is written in Markdown format, with code samples extracted from compiled, run and tested source files.

There are a few guidelines that must be adhered to when writing Play documentation.

## Markdown

All markdown files must have unique names across the entire documentation, regardless of what folders they are in. Play uses a wiki style of linking and rendering documentation.

Newline characters in the middle of paragraphs are considered hard wraps, similar to GitHub flavored markdown, and are rendered as line breaks. Paragraphs should therefore be contained on a single line.

## Links

Links to other pages in the documentation should be created using wiki markup syntax, for example:

[[Optional description|ScalaRouting]]

Images should also use the above syntax.

External links should not use the above syntax, but rather, should use the standard Markdown link syntax.

# Code samples

All supported code samples should be imported from external compiled files. The syntax for doing this is:

```
@[some-label](code/SomeFeature.scala)
```

The file should then delimit the lines that need to be extracted using `#some-label`, for example:

```
object SomeFeatureSpec extends Specification {
 "some feature" should {
 "do something" in {
 //some-label
 val msg = Seq("Hello", "world").mkString(" ")
 //some-label
 msg must_== "Hello world"
 }
 }
}
```

In the above case, the `val msg = ...` line will be extracted and rendered as code in the page. All code samples should be checked to ensure they compile, run, and if it makes sense, ensure that it does what the documentation says it does. It should not try to test the features themselves.

All code samples get run on the same classloader. Consequently they must all be well namespaced, within a package that corresponds to the part of the documentation they are associated with.

In some cases, it may not be possible for the code that should appear in the documentation to exactly match the code that you can write given the above guidelines. In particular, some code samples require the use of package names like `controllers`. As a last resort if there are no other ways around this, there are a number of directives you can put in the code to instruct the code samples extractor to modify the sample. These are:

- `###replace: foo` - Replace the next line with `foo`. You may optionally terminate this command with `###`
- `###insert: foo` - Insert `foo` before the next line. You may optionally terminate this command with `###`
- `###skip` - Skip the current line
- `###skip: n` - Skip the next n lines

For example:

```
//#controller
//###replace: package controllers
package foo.bar.controllers

import play.api.mvc._

object Application extends Controller {
 ...
}

//#controller
```

These directives must only be used as a last resort, since the point of pulling code samples out into external files is that the very code that is in the documentation is also compiled and tested. Directives break this.

It's also important to be aware of the current context of the code samples, to ensure that the appropriate import statements are documented. However it doesn't make sense to necessarily include all import statements in every code sample, so discretion must be shown here.

Guidelines for specific types of code samples are below.

## Scala

All scala code samples should be tested using specs, and the code sample, if possible, should be inside the spec. Local classes and method definitions are encouraged where appropriate. Scoping import statements within blocks are also encouraged.

## Java

All Java code samples should be tested using JUnit. Simple code samples are usually simple to include inside the JUnit test, but when the code sample is a method or a class, it gets harder. Preference should be shown to use local and inner classes, but this may not be possible, for example, a static method can only appear on a static inner class, but that means adding the static modifier to the class, which would not appear if it was an outer class. Consequently it may be necessary in some cases to pull Java code samples out into their own files.

## Scala Templates

Scala template code samples should be tested either with Specs in Scala or JUnit in Java. Note that templates are compiled with different default imports, depending on whether they live in the Scala documentation or the Java documentation. It is therefore also important to

test them in the right context, if a template is relying on Java thread locals, they should be tested from a Java action.

Where possible, template code samples should be consolidated in a single file, but this may not always be possible, for example if the code sample contains a parameter declaration.

## Routes files

Routes files should be tested either with Specs in Scala or JUnit in Java. Routes files should be named with the full package name, for example, `scalaguide.http.routing.routes`, to ensure that they are isolated from other routes code samples.

The routes compiler used by the documentation runs in a special mode that generates the reverse router inside the namespace declared by that file. This means that although a routes code sample may appear to use absolute references to classes, it is actually relative to the namespace of the router. Thus in the above routes file, if you have a route called `controllers.Application`, it will actually refer to a controller called `scalaguide.http.routing.controllers.Application`.

## SBT code

At current, SBT code samples cannot be pulled out of the documentation, since compiling and running them will require a very custom SBT setup involving using completely different classloaders and classpaths.

## Other code

Other code may or may not be testable. It may make sense to test Javascript code by running an integration test using HTMLUnit. It may make sense to test configuration by loading it. Common sense should be used here.

---

# Testing the docs

To build the docs, you'll first need to build and publish Play locally. You can do this by running `./build publishLocal` from within the `framework` directory of the playframework repository.

To ensure that the docs render correctly, run `./build run` from within the `documentation` directory. This will start a small Play server that does nothing but serve the documentation.

To ensure that the code samples compile, run and tests pass, run `./build test`.

To validate that the documentation is structurely sound, run `./build validateDocs`.

This checks that there are no broken wiki links, code references or resource links, ensures

that all documentation markdown filenames are unique, and ensures that there are no orphaned pages.

---

# Code samples from external Play modules

To avoid circular dependencies, any documentation that documents a Play module that is not a core part of Play can't include its code samples along with the rest of the Play documentation. To address this, the documentation for that module can place an entry into the `externalPlayModules` map in `project/Build.scala`, including all the extra settings (namely library dependencies) required to build the code snippets for that module.

For example:

```
val externalPlayModules: Map[String, Seq[Setting[_]]] = Map(
 "some-module" -> Seq(
 libraryDependencies += "com.example" %% "some-play-module" % "1.2.3" % "test"
,
 ...
)
```

Now place all code snippets that use that module in `code-some-module`. Now to run any SBT commands, ensuring that that module is included, run `./build -Dexternal.modules=some-module test`, or to run the tests for all modules, run `./build -Dexternal-modules=all test`.

Next: [Translating documentation](#)

# Translating the Play Documentation

Play 2.3+ provides infrastructure to aid documentation translators in translating the Play documentation and keeping it up to date.

As described in the [Documentation Guidelines](#), Play's documentation is written in markdown format with code samples extracted to external files. Play allows the markdown components of the documentation to be translated, while allowing the original code samples from the English documentation to be included in the translated documentation. This assists translators in maintaining translation quality - the code samples are kept up to date as part of the core Play project, while the translated descriptions have to be maintained manually.

In addition to this, Play also provides facilities for validating the integrity of translated documentation. This includes validating all internal links, including links to code snippets, in the translation.

---

## Prerequisites

You need to have `activator` or `sbt` installed. It will also be very useful to have a clone of the Play repository, with the branch that you're translating checked out, so that you have something to copy to start with.

If you're translating an unreleased version of the Play documentation, then you'll need to build that version of Play and publish it locally on your machine first. This can be done by running:

```
./build publishLocal
in the framework directory of the Play project.
```

## Setting up a translation

Create a new SBT project with the following structure:

```
translation-project
|- manual
| | - javaGuide
| | - scalaGuide
| | - gettingStarted
| | - etc...
|- project
| | - build.properties
| | - plugins.sbt
` - build.sbt
```

`build.properties` should contain the SBT version, ie:

```
sbt.version=0.13.8
```

`plugins.sbt` should include the Play docs sbt plugin, ie:

```
addSbtPlugin("com.typesafe.play" % "play-docs-sbt-plugin" % "2.4.x")
```

Finally, `build.sbt` should enable the Play docs plugin, ie:

```
lazy val root = (project in file(".")).enablePlugins(PlayDocsPlugin)
```

Now you're ready to start translating!

---

## Translating documentation

First off, start the documentation server. The documentation server will serve your documentation up for you so you can see what it looks like as you're going. To do this you'll need `sbt` or `activator` installed, either one is fine, in the examples here we'll be using `sbt`:

```
$ sbt run
[info] Set current project to root (in build file:/Users/jroper/tmp/foo-translation/)
[info] play - Application started (Dev)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

[Documentation](#) server started, you can now view the docs by going to <http://0:0:0:0:0:0:9000>

Now open <http://localhost:9000> in your browser. You should be able to see the default Play documentation. It's time to translate your first page.

Copy a markdown page from the Play repository into your project. It is important to ensure that the directory structure in your project matches the directory in Play, this will ensure that the code samples work.

For example, if you choose to start

with `manual/scalaGuide/main/http/ScalaActions.md`, then you need to ensure that it is in `manual/scalaGuide/main/http/ScalaActions.md` in your project.

**Note:** It may be tempting to start by copying the entire Play manual into your project. If you do do this, make sure you only copy the markdown files, that you don't copy the code samples as well. If you copy the code samples, they will override the code samples from Play, and you will lose the benefit of having those code samples automatically maintained for you.

Now you can start translating the file.

## Dealing with code samples

The Play documentation is full of code samples. As described in the [Documentation Guidelines](#), these code samples live outside of the markdown documentation and live in compiled and tested source files. Snippets from these files get included in the documentation using the following syntax:

```
@[label](code/path/to/SourceFile.java)
```

Generally, you will want to leave these snippets as is in your translation, this will ensure that the code snippets your translation stays up to date with Play.

In some situations, it may make sense to override them. You can either do this by putting the code directly in the documentation, using a fenced block, or by extracting them into your projects own compile code samples. If you do that, checkout the Play documentation sbt build files for how you might setup SBT to compile them.

---

# Validating the documentation

The Play docs sbt plugin provides a documentation validation task that runs some simple tests over the documentation, to ensure the integrity of links and code sample references. You can run this by running:

```
sbt validateDocs
```

You can also validate the links to external sites in Play's documentation. This is a separate task because it's dependent on many sites on the internet that Play's documentation links to, and the validation task in fact actually triggers DDoS filters on some sites. To run it, run:

```
sbt validateExternalLinks
```

## Translation report

Another very helpful tool provided by Play is a translation report, which shows which files have not been translated, and also tries to detect issues, for example, if the translation introduces new files, or if the translation is missing code samples. This can particularly help when translating a new version of the documentation, since the addition or removal of code samples will often be a good signal that something has changed.

To view the translation report, run the documentation server (like normal), and then visit<http://localhost:9000/@report> in your browser. By default it will serve a cached version of the report if it has been generated in the past, you can rerun the report by clicking the rerun report link.

## Deploying documentation to playframework.com

[playframework.com](#) serves documentation out of git repositories. If you want your translation to be served from playframework.com, you'll need to put your documentation into a GitHub repository, and contact the Play team to have them add it to playframework.com.

The git repository needs to be in a very particular format. The current master branch is for the documentation of the latest development version of Play. Documentation for stable versions of Play must be in branches such as 2.3.x. Documentation specific to a particular release of Play will be served from a tag of the repository with that name, for example, 2.3.1.

Once the Play team has configured playframework.com to serve your translation, any changes pushed to your GitHub repository will be picked up within about 10 minutes, as playframework.com does a `git fetch` on all repos it uses once every 10 minutes.

---

# Specifying the documentation version

By default, the `play-docs-sbt-plugin` uses the same version of the Play documentation code samples and fallback markdown files as itself, so if in `plugins.sbt` you're using `2.4.0`, when you run the documentation, you will get `2.4.0` of the documentation code samples. You can control this version by setting `PlayDocsKeys.docsVersion` in `build.sbt`:

```
PlayDocsKeys.docsVersion := "2.3.1"
```

This is particularly useful if you are wanting to provide documentation for versions of Play prior to when the `play-docs-sbt-plugin` was introduced, as far back as `2.2.0`. For `2.1.x` and earlier, the documentation was not packaged and published as a jar file, so the tooling will not work for those older versions.

Next: [Working with git](#)

# Working with Git

This guide is designed to help new contributors get started with Play. Some of the things mentioned here are conventions that we think are good and make contributing to Play easier, but they are certainly not prescriptive, you should use what works best for you.

---

## Git remotes

We recommend the convention of calling the remote for the official Play repository `origin`, and the remote for your fork your username. This convention works well when sharing code between multiple forks, and is the convention we'll use for all the remaining git commands in this guide. It is also the convention that works best out of the box with the [GitHub command line tool](#).

---

## Branches

Typically all work should be done in branches. If you do work directly on master, then you can only submit one pull request at a time, since if you try to submit a second from master, the second will contain commits from both your first and your second. Working in branches allows you to isolate pull requests from each other.

It's up to you what you call your branches, some people like to include issue numbers in their branch name, others like to use a hierarchical structure.

---

# Squashing commits

We prefer that all pull requests be a single commit. There are a few reasons for this:

- It's much easier and less error prone to backport single commits to stable branches than backport groups of commits. If the change is just in one commit, then there is no opportunity for error, either the whole change is cherry picked, or it isn't.
- We aim to have our master branch to always be releasable, not just now, but also for all points in history. If we need to back something out, we want to be confident that the commit before that is stable.
- It's much easier to get a complete picture of what happened in history when changes are self contained in one commit.

Of course, there are some situations where it's not appropriate to squash commits, this will be decided on a case by case basis, but examples of when we won't require commits to be squashed include:

- When the pull request contains commits by more than one person. In this case, we'd prefer, where it makes sense, to squash contiguous commits by the same person.
- When the pull request is coming from a fork or branch that has been shared among the community, where rewriting history will cause issues for people that have pull changes from that fork or branch.
- Where the pull request is a very large amount of work, and the commit log is useful in understanding the evolution of that work.

However, for the general case, if your pull request contains more than one commit, then you will need to squash it. To do this, or if you already have submitted a pull request and we ask you to squash it, then you should follow these steps:

1. Ensure that you have all the changes from the core master branch in your repo:

```
git fetch origin
```

2. Start an interactive rebase

```
git rebase -i origin/master
```

3. This will open up a screen in your editor, allowing you to say what should be done with each commit. If the commit message for the first commit is suitable for describing all of the commits, then leave it as is, otherwise, change the command for it from `pick` to `reword`.
4. For each remaining commit, change the command from `pick` to `fixup`. This tells git to merge that commit into the previous commit, using the commit message from the previous commit.
5. Save the file and exit your editor. Git will now start the rebase. If you told it to reword the first commit, it will prompt you in a new editor for the wording for that commit. If all goes well, then you're done, but it may be the case that there were conflicts when applying your changes to the most recent master branch. If that's the case, fix the conflicts, stage the fixes, and then run:

```
git rebase --continue
```

This may need to be repeated if there are more changes that have conflicts.

6. Now that you've rebased you can push your changes. If you've already pushed this branch before (including if you've already created the pull request), then you will have to do a force push. This can be done like so:

```
git push yourremote yourbranch --force
```

## Responding to reviews/build breakages

If your pull request doesn't pass the CI build, if we review it and ask you to update your pull request, or if for any other reason you want to update your pull request, then rather than creating a new commit, amend the existing one. This can be done by supplying the `--amend` flag when committing:

```
git commit --amend
```

After doing an amend, you'll need to do a force push using the `--force` flag:

```
git push yourremote yourbranch --force
```

# Starting over

Sometimes people find that they get their pull request completely wrong and want to start over. This is fine, however there is no need to close the original pull request and open a new one. You can use a force push to push a completely new branch into the pull request.

To start over, make sure you've got the latest changes from Play core, and create a new branch from that point:

```
git fetch origin
git checkout -b mynewbranch origin/master
```

Now make your changes, and then when you're ready to submit a pull request, assuming your old branch was called `myoldbranch`, push your new branch into that old branch in your repository:

```
git push yourremote mynewbranch:myoldbranch --force
```

Now the pull request should be updated with your new branch.

---

## A word on changing history

You may have heard it said that you shouldn't change git history once you publish it. Using `rebase` and `commit --amend` both change history, and using `push --force` will publish your changed history.

There are definitely times when git history shouldn't be changed after being published. The main times are when it's likely that other people have forked your repository, or pulled changes from your repository. Changing history in those cases will make it impossible for them to safely merge changes from your repo into their repository. For this reason, we never change history in the official Play Framework repository.

However, when it comes to your personal fork, for branches that are just intended to be pull requests, then it's a different matter - the intent of the workflow is that your changes get "published" once they get merged into the master branch. Before that, the history can be considered a work in progress.

If however your branch is a collaboration of many people, and you believe that other people have pull from your branch for good reasons, please let us know, we won't force squashing commits where it doesn't make sense.

Next: [3rd party tools](#)

## 3rd Party Tools

A big THANK YOU! to these sponsors for their support of open source projects.

---

## Continuous Integration

Our continuous integration runs on [Cloudbees](#). We not only run CI on major release and master branches, but we also perform github pull request validation using CloudBees functionality.

<https://playframework2.ci.cloudbees.com/>

---

# Profiling

We are using [YourKit](#) for profiling our Java and Scala code. YourKit really helps us keep Play's resource usage to the minimum that you'd expect.

Next: [About Play](#)

# Introducing Play 2

Since 2007, we have been working on making Java web application development easier. Play started as an internal project at Zenexity (now [Zengularity](#)) and was heavily influenced by our way of doing web projects: focusing on developer productivity, respecting web architecture, and using a fresh approach to packaging conventions from the start - breaking so-called JEE best practices where it made sense.

In 2009, we decided to share these ideas with the community as an open source project. The immediate feedback was extremely positive and the project gained a lot of traction. Today - after two years of active development - Play has several versions, an active community of more than 10,000 people, with a growing number of applications running in production all over the globe.

Opening a project to the world certainly means more feedback, but it also means discovering and learning about new use cases, requiring features and un-earthing bugs that we were not specifically considered in the original design and its assumptions. During the two years of work on Play as an open source project we have worked to fix this kind of issues, as well as to integrate new features to support a wider range of scenarios. As the project has grown, we have learned a lot from our community and from our own experience - using Play in more and more complex and varied projects.

Meanwhile, technology and the web have continued to evolve. The web has become the central point of all applications. HTML, CSS and JavaScript technologies have evolved quickly - making it almost impossible for a server-side framework to keep up. The whole web architecture is fast moving towards real-time processing, and the emerging requirements of today's project profiles mean SQL no longer works as the exclusive datastore technology. At the programming language level we've witnessed some monumental changes with several JVM languages, including Scala, gaining popularity.

That's why we created Play 2, a new web framework for a new era.

---

# Built for asynchronous programming

Today's web applications are integrating more concurrent real-time data, so web frameworks need to support a full asynchronous HTTP programming model. Play was initially designed to handle classic web applications with many short-lived requests. But now, the event model is the way to go for persistent connections - through Comet, long-polling and WebSockets.

Play 2 is architected from the start under the assumption that every request is potentially long-lived. But that's not all: we also need a powerful way to schedule and run long-running tasks. The Actor-based model is unquestionably the best model today to handle highly concurrent systems, and the best implementation of that model available for both Java and Scala is Akka - so it's going in. Play 2 provides native [Akka](#) support for Play applications, making it possible to write highly-distributed systems.

---

# Focused on type safety

One benefit of using a statically-typed programming language for writing Play applications is that the compiler can check parts of your code. This is not only useful for detecting mistakes early in the development process, but it also makes it a lot easier to work on large projects with a lot of developers involved.

Adding Scala to the mix for Play 2, we clearly benefit from even stronger compiler guarantees - but that's not enough. In Play 1.x, the template system was dynamic, based on the Groovy language, and the compiler couldn't do much for you. As a result, errors in templates could only be detected at run-time. The same goes for verification of glue code with controllers.

In version 2.0, we really wanted to push this idea of having Play check most of your code at compilation time further. This is why we decided to use the Scala-based template engine as the default for Play applications - even for developers using Java as the main programming language. This doesn't mean that you have to become a Scala expert to write templates in Play 2, just as you were not really required to know Groovy to write templates in Play 1.x.

In templates, Scala is mainly used to navigate your object graph in order to display relevant information, with a syntax that is very close to Java's. However, if you want to unleash the power of Scala to write advanced templates abstractions, you will quickly discover how Scala, being expression-oriented and functional, is a perfect fit for a template engine.

And that's not only true for the template engine: the routing system is also fully type-checked. Play 2 checks your routes' descriptions, and verifies that everything is consistent, including the reverse routing part.

A nice side effect of being fully compiled is that the templates and route files will be easier to package and reuse. You also get a significant performance gain on these parts at run-time.

---

## Native support for Java and Scala

Early in the Play project's history, we started exploring the possibility of using the Scala programming language for writing Play applications. We initially introduced this work as an external module, to be able to experiment freely without impacting the framework itself.

Properly integrating Scala into a Java-based framework is not trivial. Considering Scala's compatibility with Java, one can quickly achieve a first naive integration that simply uses Scala's syntax instead of Java's. This, however, is certainly not the optimal way of using the language. Scala is a mix of true object orientation with functional programming. Leveraging the full power of Scala requires rethinking most of the framework's APIs.

We quickly reached the limits of what we can do with Scala support as a separate module. Initial design choices we made in Play 1.x, relying heavily on Java reflection API and byte code manipulation, have made it harder to progress without completely rethinking some essential parts of Play's internals. Meanwhile, we have created several awesome components for the Scala module, such as the new type-safe template engine and the brand new SQL access component [Anorm](#). This is why we decided that, to fully unleash the power of Scala with Play, we would move Scala support from a separate module to the core of Play 2, which is designed from the beginning to natively support Scala as a programming language.

Java, on the other hand, is certainly not getting any less support from Play 2; quite the contrary. The Play 2 build provides us with an opportunity to enhance the development

experience for Java developers. Java developers get a real Java API written with all the Java specificity in mind.

---

# Powerful build system

From the beginning of the Play project, we have chosen a fresh way to run, compile and deploy Play applications. It may have looked like an esoteric design at first, but it was crucial to providing an asynchronous HTTP API instead of the standard Servlet API, short feedback cycles through live compilation and reloading of source code during development, and promoting a fresh packaging approach. Consequently, it was difficult to make Play follow the standard JEE conventions.

Today, this idea of container-less deployment is increasingly accepted in the Java world. It's a design choice that has allowed the Play framework to run natively on platforms like [Heroku](#), which introduced a model that we consider the future of Java application deployment on elastic PaaS platforms.

Existing Java build systems, however, were not flexible enough to support this new approach. Since we wanted to provide straightforward tools to run and deploy Play applications, in Play 1.x we created a collection of Python scripts to handle build and deployment tasks.

Meanwhile, developers using Play for more enterprise-scale projects, which require build process customization and integration with their existing company build systems, were a bit lost. The Python scripts we provided with Play 1.x are in no way a fully-featured build system and are not easily customizable. That's why we've decided to go for a more powerful build system for Play 2.

Since we need a modern build tool, flexible enough to support Play original conventions and able to build Java and Scala projects, we have chosen to integrate [sbt](#) in Play 2. This, however, should not scare existing Play users who are happy with the simplicity of the original Play build. We are using [Activator](#) to provide simple commands like `activator new`, `run`, `start` on top of an extensible model and if you need to change the way your application is built and deployed, the fact that a Play project is a standard sbt project gives you all the power you need to customize and adapt it.

This also means better integration with Maven projects out of the box, the ability to package and publish your project as a simple set of JAR files to any repository, and especially live compiling and reloading at development time of any depended project, even for standard Java or Scala library projects.

---

# Datastore and model integration

'Data store' is no longer synonymous with 'SQL database', and probably never was. A lot of interesting data storage models are becoming popular, providing different properties for different scenarios. For this reason it has become difficult for a web framework like Play to make bold assumptions regarding the kind of data store that developers will use. A generic model concept in Play no longer makes sense, since it is almost impossible to abstract over all these kinds of technologies with a single API.

In Play 2, we wanted to make it really easy to use any data store driver, ORM, or any other database access library without any special integration with the web framework. We simply want to offer a minimal set of helpers to handle common technical issues, like managing the connection bounds. We also want, however, to maintain the full-stack aspect of Play framework by bundling default tools to access classical databases for users WHO don't have specialized needs, and that's why Play 2 comes with built-in relational database access libraries such as [Ebean](#), JPA and Anorm.

Next: [Play user groups](#)

## Play User Groups

---

### New York

<http://www.meetup.com/Play-NYC/>

### Berlin

<http://www.meetup.com/Play-Berlin-Brandenburg/>

### Cologne

Scala User Group Köln / Bonn

(We also talk and do presentations about Play)

[Xing](#) / [Twitter](#)

### Buenos Aires

<http://www.meetup.com/play-argentina/>

---

## Stockholm

<http://www.meetup.com/play-stockholm/>

---

## Belgium

<http://www.play-be.org>

---

## Japan

- <http://www.playframework-ja.org/>
  - [https://groups.google.com/forum/?fromgroups#!forum/play\\_ja](https://groups.google.com/forum/?fromgroups#!forum/play_ja)
- 

## Republic of Korea

*Korea Play! User Group*

- Facebook
  - Github
  - Slack
- 

## New Delhi - INDIA

<http://www.meetup.com/Reactive-Application-Programmers-in-Delhi-NCR/>

Next: Play releases

Next: Play releases

## What's new in Play 2.4

This page highlights the new features of Play 2.4. If you want learn about the changes you need to make to migrate to Play 2.4, check out the [Play 2.4 Migration Guide](#).

---

## Dependency Injection

Play now supports dependency injection out of the box.

## Motivation

A long term strategy for Play is to remove Play's dependence on global state. Play currently stores a reference to the current application in a static variable, and then uses this variable in many places throughout its codebase. Removing this has the following advantages:

- Applications become easier to test and components become easier to mock.
- More interesting deployment scenarios are possible, such as multiple Play instances in a single JVM, or embedding a lightweight Play application.
- The application lifecycle becomes easier to follow and reason about.

Removing Play's global state is however a big task that will require some disruptive changes to the way Play applications are written. The approach we are taking to do this is to do as much as possible in Play 2.4 while maintaining backwards compatibility. For a time, many of Play's APIs will support both methods that rely on require global state and methods that don't rely on global state, allowing you to migrate your application to not depend on global state incrementally, rather than all at once when you upgrade to Play 2.4.

The first step to removing global state is to make it such that Play components have their dependencies provided to them, rather than looking them up statically. This means providing out of the box support for dependency injection.

## Approach

In the Java ecosystem, the approach to dependency injection is generally well agreed upon in [JSR 330](#), but the right implementation is widely debated, with many existing competing implementations such as Guice, Spring and JEE itself.

In the Scala ecosystem, the approach to dependency injection is not generally agreed upon, with many competing compile time and runtime dependency injection approaches out there.

Play's philosophy in providing a dependency injection solution is to be unopinionated in what approaches we allow, but to be opinionated to the approach that we document and provide out of the box. For this reason, we have provided the following:

- An implementation that uses [Guice](#) out of the box
- An abstraction that allows other JSR 330 implementations to be plugged in
- All Play components can be instantiated using plain constructors or factory methods
- Traits that instantiate Play components that can be mixed together in a cake pattern like style to assist with compile time dependency injection

You can read more about Play's dependency injection support for [Java](#) and [Scala](#).

---

# Testing

One of the biggest advantages of introducing dependency injection to Play is that many parts of Play can now be much easier to test. Play now provides a number of APIs to assist in mocking and overriding components, as well as being able to test interactions with Play components in isolation from the rest of your Play application.

You can read about these new APIs here:

- Configuring Guice components in [Java](#) and [Scala](#)
  - Testing database access code in [Java](#) and [Scala](#)
  - Testing web service client code in [Java](#) and [Scala](#)
- 

## Embedding Play

It is now straightforward to embed a Play application. Play 2.4 provides both APIs to start and stop a Play server, as well as routing DSLs for Java and Scala so that routes can be embedded directly in code.

In Java, see [Embedding Play](#) as well as information about the [Routing DSL](#).

In Scala, see [Embedding Play](#) as well as information about the [String Interpolating Routing DSL](#).

---

## Aggregated reverse routers

Play now supports aggregating reverse routers from multiple sub projects into a single shared project, with no dependency on the project the routes files came from. This allows a modular Play application to use the Play reverse router as an API between modules, allowing them to render URLs to each other without depending on each other. It also means a dependency free reverse router could be extracted out of a Play project, and published, for use by external projects that invoke the APIs provided by the project.

For details on how to configure this, see [Aggregating Reverse Routers](#).

---

## Java 8 support

Play 2.4 now requires JDK 8. Due to this, Play can, out of the box, provide support for Java 8 data types. For example, Play's JSON APIs now support Java 8 temporal types including `Instance`, `LocalDateTime` and `LocalDate`.

The Play documentation now shows code examples using Java 8 syntax for anonymous inner classes. As an example, here's how some of the code samples have changed:

Before:

```
return promise(new Function0<Integer>() {
 public Integer apply() {
 return longComputation();
 }
}).map(new Function<Integer,Result>() {
 public Result apply(Integer i) {
 return ok("Got " + i);
 }
});
```

After:

```
return promise(() -> longComputation())
 .map((Integer i) -> ok("Got " + i));
```

## Maven/sbt standard layout

Play will now let you use either its default layout or the directory layout that is the default for Maven and SBT projects. See the [Anatomy of a Play application](#) page for more details.

## Anorm

Anorm has been extracted into a separate project with its own lifecycle, allowing anorm to move at its own pace, not bound to Play. The anorm project can be found [here](#).

New features in anorm include:

- New positional getter on `Row`.
- Unified column resolution by label, whatever it is (name or alias).
- New streaming API; Functions `fold` and `foldWhile` to work with result stream (e.g. `SQL("Select count(*) as c from Country").fold(01) { (c, _) => c + 1 }`). Function `withResult` to provide custom stream parser (e.g. `SQL("Select name from Books").withResult(customTailrecParser(_, List.empty[String]))`).
- Supports array (`java.sql.Array`) from column (e.g. `SQL("SELECT str_arr FROM tbl").as(scalar[Array[String]].*)`) or as parameter (e.g. `SQL""""UPDATE Test SET langs = ${Array("fr", "en", "ja") }""".execute()`).
- Improved conversions for numeric and boolean columns.
- New conversions for binary columns (bytes, stream, blob), to parsed them as `Array[Byte]` or `InputStream`.
- New conversions for Joda `Instant` or `DateTime`, from `Long`, `Date` or `Timestamp` column.
- Added conversions to support `List[T]`, `Set[T]`, `SortedSet[T]`, `Stream[T]` and `Vector[T]` as multi-value parameter.

- New conversion to parse text column as UUID (e.g. `SQL("SELECT uuid_as_text").as(scalar[java.util.UUID].single())`).
- 

## Ebean

Play's Ebean support has been extracted into a separate project with its own lifecycle, allowing Ebean support to move at its own pace, not bound to Play. The play-ebean project can be found [here](#).

play-ebean now supports Ebean 4.x.

---

## HikariCP

[HikariCP](#) is now the default JDBC connection pool. Its properties can be directly configured using `.conf` files and you should rename the configuration properties to match what is expected by HikariCP.

---

## WS

WS now supports Server Name Indication (SNI) in HTTPS – this solves a number of problems with HTTPS based CDNs such as Cloudflare which depend heavily on SNI.

---

## Experimental Features

Play provides two new experimental features. These are labelled as experimental because the APIs for them have not yet been finalised, and may change from one release to the next. Binary compatibility is not guaranteed on these APIs.

### Akka HTTP support

Play supports a new Akka HTTP backend, as an alternative to the current Netty backend. For instructions on using it, see [Akka Http Server](#).

### Reactive Streams Support

Play provides an iteratees based implementation of [Reactive Streams](#), allowing other Reactive Streams implementations, such as Akka Streams or RxJava, to be used with Play's iteratee IO APIs. For more information, see [Reactive Streams Integration](#).

Next: [What's new in Play 2.3?](#)

# Play Modules

This is a temporary location for listing modules, until there will be a location to register them on the Play website.

Blog posts describing the process of writing a module can be found on [objectify.be](#).

## Airbrake.io notifier

- Website: <http://teamon.eu/play-airbrake/>
  - Documentation: <https://github.com/teamon/play-airbrake/blob/master/README.md>
  - Short description: Send exception notifications to airbrake.io
- 

## Amazon SES module (Scala)

- Website: <https://github.com/Rhinofly/play-mailer>
  - Documentation: <https://github.com/Rhinofly/play-mailer/blob/master/README.md>
  - Short description: SES (Simple Email Service) API wrapper for Play
- 

## Amazon S3 module (Scala)

- Website: <https://github.com/Rhinofly/play-s3>
  - Documentation: <https://github.com/Rhinofly/play-s3/blob/master/README.md>
  - Short description: S3 (Simple Storage Service) API wrapper for Play
- 

## Amf module (Scala)

- Website: <https://github.com/Rhinofly/play-amf>
  - Documentation: <https://github.com/Rhinofly/play-amf/blob/master/README.md>
  - Short description: AMF (ActionScript Message Format) support for Play
- 

## Authentication and Authorization module (Scala)

- Website: <https://github.com/t2v/play20-auth>
- Documentation(en): <https://github.com/t2v/play20-auth/blob/master/README.md>
- Documentation(ja): <https://github.com/t2v/play20-auth/blob/master/README.ja.md>
- Short description This module provides an authentication and authorization way
- Now also has 2.1-SNAPSHOT support: <https://github.com/t2v/play20-auth/tree/play21>

---

# Authenticity Token module

- **Website:** <https://github.com/orefalo/play2-authenticitytoken>
  - **Documentation:** <https://github.com/orefalo/play2-authenticitytoken/blob/master/README.md>
  - **Short description:** Brings back play1 authenticity token - provides a way around CSRF attacks
- 

# ClojureScript Plugin

- **Website:** <https://bitbucket.org/jmhofer/play-clojurescript> (docs, code)
  - **Short description:** Compiles ClojureScript asset files to JavaScript
- 

# Cloudfront module (Scala)

- **Website:** <https://github.com/mchv/play2-cloudfront>
  - **Documentation:** <https://github.com/mchv/play2-cloudfront/blob/master/README.md>
  - **Short description:** This module helps to integrate a play application with Cloudfront CDN.
- 

# Currency Converter (Java)

- **Website:** <https://edulify.github.io/play-currency-converter-module.edulify.com/>
  - **Documentation:** <https://github.com/edulify/play-currency-converter-module.edulify.com/blob/master/README.md>
  - **Short description:** Currency converter for Play. It uses web services to get the current exchange rate.
- 

# Deadbolt 2 Plugin

- **Website (docs, sample):** <https://github.com/schaloner/deadbolt-2>
  - **Short description:** Deadbolt is an authorisation mechanism for defining access rights to certain controller methods or parts of a view using a simple AND/OR/NOT syntax
- 

# DDSL Plugin - Dynamic Distributed Service Locator

- **Website (docs, sample):** <https://github.com/mbknor/ddsl-play2-module>
  - **Short description:** DDSL - Dynamic Distributed Service Locator - Makes it really easy to create your own dynamic “private cloud” in-house or on EC2/Joyent etc using ZooKeeper
- 

# Dust Plugin

- **Website (docs, sample):** <https://github.com/typesafehub/play-plugins/tree/master/dust>
- **Short description:** Provides support for the dust client side template language

---

# Google Closure Template Plugin

- **Website (docs, sample):** <https://github.com/gawkermedia/play2-closure>
  - **Short description:** Provides support for Google Closure Templates
- 

## Elasticsearch

- **Website:** <https://github.com/cleverage/play2-elasticsearch>
  - **Documentation:** <https://github.com/cleverage/play2-elasticsearch/blob/master/README.md>
  - **Repository:** <http://cleverage.github.io/play2-elasticsearch/releases/>
  - **Short description** Indexing/Requesting Object in Embedded ElasticSearch Server or remote(s) Node(s).
- 

## Ember.js

- **Website:** <https://github.com/krumpi/play-emberjs>
  - **Documentation:** <https://github.com/krumpi/play-emberjs/blob/master/README.md>
  - **Short description** Supports precompilation of ember.js/handlebars templates.
- 

## funcy - Page Driven Functional Tests (Java)

- **Website:** <https://github.com/joergviola/funcy>
  - **Documentation:** <https://github.com/joergviola/funcy/blob/master/README.md>
  - **Repository:** <http://www.joergviola.de/releases/>
  - **Short description** Simplifies writing functional test using Page Driver classes.
- 

## FolderMessages plugin

- **Website:** <https://github.com/germanosin/play-foldermessages>
  - **Short Description:** Allows you to split localization messages files into separate manageable files.
- 

## Flyway plugin

- **Website:** <https://github.com/tototoshi/play-flyway>
  - **Documentation:** <https://github.com/tototoshi/play-flyway/blob/master/README.md>
  - **Short Description:** Supports database migration with Flyway.
- 

## Geolocation (Java)

- **Website:** <https://edulify.github.io/play-geolocation-module.edulify.com/>

- **Documentation:** <https://github.com/edulify/play-geolocation-module.edulify.com/blob/master/README.md>
  - **Short description:** Module to retrieve Geolocation data based on IP.
- 

## Google's HTML Compressor (Java and Scala)

- **Website:** <https://github.com/mohiva/play-html-compressor>
  - **Documentation:** <https://github.com/mohiva/play-html-compressor/blob/master/README.md>
  - **Short description:** Google's HTML Compressor for Play 2.
- 

## Groovy Templates plugin

- **Website:** <https://github.com/manuelbernhardt/play2-groovy-templates>
  - **Documentation:** <https://github.com/manuelbernhardt/play2-groovy-templates/blob/master/README.md>
  - **Short description:** This module brings the Groovy templates engine of Play 1 to Play 2.
- 

## Groovy Templates plugin - gt-engine-play2

- **Website:** <https://github.com/mbknor/gt-engine-play2>
  - **Documentation:** <https://github.com/mbknor/gt-engine-play2/blob/master/README.markdown>
  - **Short description:** This module brings the Groovy Template engine from play 1 to Play. It uses gt-engine which is used by the Play 1 module "Faster Groovy Templates"<http://www.playframework.com/modules/fastergt>
  - **Samples:** <https://github.com/mbknor/gt-engine-play2/tree/master/samples>
- 

## Guice Plugin (Java and Scala)

- **Website (docs, sample):** <https://github.com/typesafehub/play-plugins/tree/master/guice>
  - **Short description:** Provides DI via Guice
- 

## HTML5 Tags module (Java and Scala)

- **Website:** <https://github.com/loicdescotte/Play2-HTML5Tags>
- **Documentation:** <https://github.com/loicdescotte/Play2-HTML5Tags/blob/master/README.md>
- **Short description:** These tags add client side validation capabilities, based on model constraints (e.g required, email pattern, max|min length...) and specific input fields (date, telephone number, url...) to Play templates

---

# InputValidator (Scala)

- **Website:** <https://github.com/seratch/inputvalidator/tree/master/play-module>
  - **Short description:** Provides a simple validation API for Play
- 

# JackRabbit Plugin (Java and Scala)

- **Website (docs, sample):** [JackRabbit Plugin GitHub](#)
  - **Short description:** Apache JackRabbit Plugin for play! framework 2
- 

# Japid module

- **Website:** <https://github.com/branaway/japid42>
  - **Documentation:** <https://github.com/branaway/japid42>
  - **Short description:** This module provides Japid java templates for Play
- 

# JsMessages

- **Website:** <https://github.com/julienrf/play-jsmessages>
  - **Short description:** Allows to compute localized messages on client side.
- 

# JSON minification Plugin

- **Website:** <https://github.com/joscha/play-jsonminify>
  - **Documentation:** <https://github.com/joscha/play-jsonminify/blob/master/README.md>
  - **Short description:** Allows JSON asset pretty-printing and minification
- 

# JSONP filter

- **Website:** <https://github.com/julienrf/play-jsonp-filter>
  - **Short description:** Enables JSONP on your existing HTTP API.
- 

# Lessc Plugin

- **Website:** <https://github.com/jmparsons/play-lessc>
  - **Documentation:** <https://github.com/jmparsons/play-lessc/blob/master/README.md>
  - **Short description:** Allows [Less](#) command line compilation through Node
- 

# Liquibase Module

- **Website:** <https://github.com/Ticketfly/play-liquibase>
  - **Documentation:** <https://github.com/Ticketfly/play-liquibase/blob/master/README.md>
  - **Short description:** Runs [Liquibase](#) database schema migrations on app startup
- 

## Manual Dependency Injection Plugin (Java and Scala)

- **Website (docs, sample):** <https://github.com/typesafehub/play-plugins>
  - **Short description:** Provides DI via manual injection
- 

## Memcached Plugin

- **Website:** <https://github.com/mumoshu/play2-memcached>
  - **Short description:** Provides a memcached based cache implementation
- 

## Messages Compiler Plugin (Scala)

- **Website:** <https://github.com/tegonaL/play-messagescompiler>
  - **Documentation:** <https://github.com/tegonaL/play-messagescompiler/blob/master/readme.md>
  - **Short description:** Provides type safety for the project's messages.
- 

## MongoDB Jackson Mapper Plugin (Java)

- **Website (docs, sample):** <https://github.com/vznet/play-mongo-jackson-mapper>
  - **Short description:** Provides managed MongoDB access and object mapping using Jackson annotations
- 

## MongoDB Jongo Plugin (Java)

- **Website (docs, sample):** <https://github.com/alexanderjaryis/play-jongo>
  - **Short description:** Provides managed MongoDB access and object mapping using [Jongo](#)
- 

## MongoDB Morphia Plugin (Java)

- **Website (docs, sample):** <https://github.com/leodagdag/play2-morphia-plugin>
- **Short description:** Provides managed MongoDB access and object mapping via Morphia

---

# MongoDB Salat, Casbah Plugin (Scala)

- **Website (docs, sample):** <https://github.com/leon/play-salat>
  - **Short description:** Provides managed MongoDB access and object mapping using Salat and Casbah
- 

## Mountable routing

- **Website:** <http://teamon.eu/play-navigator/>
  - **Documentation:** <https://github.com/teamon/play-navigator/blob/master/README.md>
  - **Description:** <http://codetunes.com/2012/05/09/scala-dsl-tutorial-writing-web-framework-router>
- 

## Mustache (Java, Scala)

- **Website:** <https://github.com/julienba/play2-mustache>
  - **Documentation:** <https://github.com/julienba/play2-mustache>
  - **Short description:** Mustache template support
- 

## Native Packaging Module

- **Website:** <https://github.com/kryptt/play2-native-packager-plugin>
  - **Documentation:** <https://github.com/kryptt/play2-native-packager-plugin/blob/master/README.md>
  - **Short description:** Allow to package Play! 2.x applications as a standard system package (deb/rpm/msi).
- 

## NINA (Scala)

- **Website (docs, sample):** <https://github.com/dontcare4free/nina>
  - **Short description:** Provides a typesafe way to query SQL databases with a special emphasis on selective querying
- 

## Origami: OrientDB O/G Mapper (Java and Scala)

- **Website (docs, sample):** [Origami Plugin GitHub](#)
  - **Short description:** Origami plugin is a Java O/G mapper for the OrientDB with Play! Framework 2.
- 

## PDF module (Java)

- **Website:** <https://github.com/innoveit/play2-pdf>

- **Documentation:** <https://github.com/innoveit/play2-pdf/blob/master/README.md>
  - **Short description:** Generate PDF output from HTML templates
- 

## Play! Authenticate (Java)

- **Website:** <https://joscha.github.io/play-authenticate/>
  - **Documentation:** <https://github.com/joscha/play-authenticate/blob/master/README.md>
  - **Short description:** A highly customizable authentication module for Play
- 

## play2-sprites

- **Website:** <https://github.com/koofr/play2-sprites/>
  - **Short description:** play2-sprites is an sbt plugin that generates sprites from images.
- 

## Play-Bootstrap3 (Java and Scala)

- **Website:** <http://play-bootstrap3.herokuapp.com/>
  - **Documentation:** <http://play-bootstrap3.herokuapp.com/docs>
  - **Repository:** <https://github.com/adrianhurt/play-bootstrap3>
  - **Short description:** a collection of input helpers and field constructors for Play Framework 2.4 to render Twitter Bootstrap 3 HTML code.
- 

## play-jaxrs (Java)

- **Website (docs, sample):** [play-jaxrs](#)
  - **Short description:** a JAX-RS router plugin for play java apps
- 

## Play-pac4j (Java and Scala)

- **Website:** <https://github.com/leleuj/play-pac4j>
  - **Documentation:** <https://github.com/leleuj/play-pac4j/blob/master/README.md>
  - **Short description:** Play client in Scala and Java which supports OAuth/CAS/OpenID/HTTP authentication and user profile retrieval
- 

## Play PlovPlugin

- **Website (docs, sample):** <https://github.com/benmccann/play-plov-plugin>
  - **Short description:** Adds Closure Compiler and Closure Library support to Play
- 

## Play-Slick

- **Website (docs, sample):** <https://github.com/freekh/play-slick>
- **Short description:** This plugin makes Slick a first-class citizen of Play.

---

## Pusher

- **Website:** <https://pusher.com/>
  - **Documentation:** <https://github.com/tindr/Play2Pusher>
  - **Short description:** Easily interact with the Pusher Service within your Play application.
- 

## Play Dok

- **Website:** <http://fudok.com/>
  - **Documentation:** <https://github.com/cchantep/play-dok/>
  - **Short description:** Library to integrate Fukdok PDF templating service with your Play application.
- 

## Qunit (Java)

- **Website:** <https://github.com/gcusnieux/play20-qunit>
  - **Documentation:** <https://github.com/gcusnieux/play20-qunit>
  - **Short description:** JavaScript unit test suite
- 

## Redis Plugin (Java and Scala)

- **Website (docs, sample):** <https://github.com/typesafehub/play-plugins>
  - **Short description:** Provides a redis based cache implementation, also lets you use Redis specific APIs
- 

## Swaggerkit (Scala)

- **Website (docs, sample, code):** <https://github.com/eamalink/swaggerkit>
  - **Short description:** Helps you expose a [Swagger](#) specification of a JSON REST API built with Play in a clean way.
- 

## Emailer Plugin (Java and Scala)

- **Website (docs, sample):** <https://github.com/playframework/play-mailer>
  - **Short description:** Provides an emailer based on apache commons-email
- 

## Roy Compiled Asset Plugin (Ray)

- **Website:** <https://github.com/pufuwuzu/ray>
  - **Blog post:** <http://brianmckenna.org/blog/ray>
  - **Short description:** Compiles [Roy](#) files to JavaScript
- 

## Sass Plugin

- **Website:** <https://github.com/jlitola/play-sass>
  - **Short description:** Asset handling for [Sass](#) files
- 

## ScalikeJDBC Plugin (Scala)

- **Website:** <https://github.com/scalikejdbc/scalikejdbc-play-support>
  - **Short description:** Provides yet another database access API for Play
- 

## SecureSocial (Java and Scala)

- **Website:** <http://securesocial.ws/>
  - **Short description:** An authentication module supporting OAuth, OAuth2, OpenID, Username/Password and custom authentication schemes.
- 

## Silhouette (Scala)

- **Website:** <http://silhouette.mohiva.com/>
  - **Documentation:** <http://silhouette.mohiva.com/docs/>
  - **Short description:** An authentication library that supports several authentication methods, including OAuth1, OAuth2, OpenID, Credentials, Basic Authentication, Two Factor Authentication or custom authentication schemes.
- 

## Sitemap Generator (Java)

- **Website:** <https://edulify.github.io/play-sitemap-module.edulify.com/>
  - **Documentation:** <https://github.com/edulify/play-sitemap-module.edulify.com/blob/master/README.md>
  - **Short description:** Automatic [sitemaps](#) generator for Play
- 

## Snapshot Plugin (Java and Scala)

- **Website (docs, sample):** <https://github.com/vznet/play-snapshot>
  - **Short description:** Provides a hash bang snapshot functionality using HtmlUnit, as described by Google [here](#)
- 

## socket.io.play (scala only, pre-alpha)

- **Website:** <https://github.com/milliondreams/socket.io.play>
- **Documentation:** <https://github.com/milliondreams/socket.io.play/blob/master/README.md>

---

# Stateless client authentication (Scala)

- Website: <https://github.com/blendlabs/play20-stateless-auth>
  - Documentation: <https://github.com/blendlabs/play20-stateless-auth/blob/master/README.md>
  - Short description: Provides required and optional authentication without requiring server-side state (signed auth data is stored on the client)
- 

# Statsd Plugin (Java and Scala)

- Website (docs, sample): <https://github.com/vznet/play-statsd>
  - Short description: Provides a statsd client
- 

# Stylus Plugin

- Website: <https://github.com/patiencelabs/play-stylus>
  - Short description: Support for [Stylus](#) CSS compilation
- 

# TinkerPop Frames O/G Mapper Plugins (Java)

- Website (docs, sample): GitHub: [Frames-Neo4j Plugin](#) / [Frames-OrientDB Plugin](#) / [Frames-Titan Plugin](#)
  - Short description: Java O/G mapper plugins for GraphDBs
- 

# Typesafe util Plugin (Scala)

- Website (docs, sample): <https://github.com/typesafehub/play-plugins>
  - Short description: Provides request header based security and syntactic sugar to deal with plugins
- 

# Typesafe SbtGoodies Plugin

- Website (docs, sample): <https://github.com/typesafehub/play-plugins/tree/master/sbtgoodies>
  - Short description: Provides extra sbt commands
- 

# TypeScript Plugin

- Website: <https://github.com/mumoshu/play2-typescript>
- Short description: Asset handling for [TypeScript](#) files

---

# WAR Module

- **Website:** <https://github.com/dlecan/play2-war-plugin>
  - **Documentation:** <https://github.com/dlecan/play2-war-plugin/blob/develop/README.md>
  - **Short description:** Allow to package Play! 2.x applications into standard WAR packages.
- 

# XForward module

- **Website:** <https://github.com/olsego/play2-xforward>
  - **Documentation:** <https://github.com/olsego/play2-xforward/blob/master/Readme.md>
  - **Short description:** This module brings back the missing Proxy forwarding settings from Play1.
- 

# XWiki Rendering module (Scala)

- **Website:** <https://literalice.github.io/play-xwiki-rendering/>
  - **Documentation:** <https://github.com/literalice/play-xwiki-rendering/>
  - **Short description:** XWiki Rendering Framework integration for Play
- 

# Thymeleaf module (Scala)

- **Website:** <https://github.com/dmitraver/scala-play-thymeleaf-plugin>
- **Documentation:** <https://github.com/dmitraver/scala-play-thymeleaf-plugin/blob/master/README.md>
- **Short description:** Allows to use [Thymeleaf](#) template engine as an alternative to Twirl