

**HCTM Application Security Standards for the Play framework**

***Version 0.07***

1 Background 5

2 Document Structure 5

3 Standards and Play framework 5

4 Technical Guidance 10

4.1 Authentication 10

4.1.1 Authentication controls shall be enforced on the server side 10

4.1.2 All protected resources must be authenticated before access 11

4.1.3 Applications shall respond with a generic error message when users enter the wrong username/password combination 12

4.1.4 Account lockout mechanisms shall be implemented 12

4.1.5 PII handled by the application shall traverse encrypted links 12

4.1.6 Account passwords shall be stored in protected form at the server-side 13

4.1.7 Account passwords shall never be stored at the client-side in insecure storage 13

4.1.8 All authentication credentials for accessing services external to the application shall be encrypted 14

4.1.9 Users shall be required to re-authenticate to the web application before executing any sensitive transaction 14

4.2 Session Management 15

4.2.1 The session IDs shall be at least 128 bits and random 15

4.2.2 Sessions shall be invalidated 15

4.2.3 Cookies attributes 16

4.3 Access Control 17

4.3.1 Access Control shall be enforced server-side 17

4.3.2 Access control shall be enforced for all resources 17

4.3.3 Directory browsing shall be disabled 18

4.3.4 The application shall generate strong random anti-CSRF tokens 18

4.4 Input Validation 19

4.4.1 All input validation shall be performed on the server side 19

4.4.2 The application shall use controls to avoid injection vulnerabilities 22

4.4.3 All untrusted data that is output to the user shall be properly escaped/encoded 24

4.4.4 Input handling libraries shall be configured to use secure-defaults 24

4.4.5 The application shall use implementation-language-level types 24

4.5 Error Handling and Logging 24

4.5.1 The application shall maintain a separate security log 24

4.5.2 The application error pages shall not display sensitive information 26

4.6 Cryptography and Data Protection 27

4.6.1 All cryptographic functions used to protect server-side application secrets shall be implemented server side, preferably by a central component 27

4.7 Secure Communication 27

4.7.1 TLS shall be used for all connections 27

4.8 HTTP Security 27

4.8.1 The application shall only accept a defined set of HTTP request methods 27

4.8.2 Every HTTP response shall contain a content type header specifying a safe character set 28

4.8.3 The HTTP header, ‘*X-Frame-Options*’ and X-Content-Type-Options shall be used 28

4.8.4 Headers shall not contain sensitive information 28

Document Revision History

| Version | Modification | Date | Author |
| --- | --- | --- | --- |
| 0.01 | Document Creation | October 1, 2014 | Koen Buyens |
| 0.01 | Document Review | October 20, 2014 | Chandu Ketkar |
| 0.02 | Document Edits | October 22, 2014 | Koen Buyens |
| 0.02 | Document Review | October 29, 2014 | Ugo Enyioha |
| 0.03 | Document Edits | October 30, 2014 | Koen Buyens |
| 0.03 | Document Review | November 8, 2014 | Ugo Enyioha |
| 0.04 | Document Edits | November 11, 2014 | Koen Buyens |
| 0.05 | Document Edits | November 24, 2014 | Koen Buyens |
| 0.05 | Document Review | November 26, 2014 | Chandu Ketkar |
| 0.06 | Document Edits | November 26, 2014 | Chandu Ketkar |
| 0.06 | Document Review | November 26, 2014 | Carl Schwarz |
| 0.07 | Document Edits | December 1, 2014 | Koen Buyens |

# Background

This document contains security standards and best practices for the Play framework. These standards need to be adhered to while designing and implementing security features using the framework.

The technical guidance is sourced from various community sources, such as the Play framework documentation[[1]](#footnote-1) and code[[2]](#footnote-2), Cigital’s expertise, and informed by Bosch’s security context – including the types of applications issues Cigital has repeatedly discovered during HCTM application assessments.

# Document Structure

The technical guidance is categorized by security concerns, such as authentication, authorization, and session management. For each of these security concerns, the guide elaborates on how to implement the security features mandated by the HCTM Application Security Standards[[3]](#footnote-3) document in a secure way within the Play framework context.

The technical guidance follows a three-step approach:

1. Use Play framework-provided features whenever available. This step includes secure configuration of these features.
2. Implement custom features, but use Play framework-provided mechanisms to wire the functionality into the application.
3. Implement custom features without using any Play framework functionality.

# Standards and Play framework

This section documents whether the Play framework supports each of the prescribed standards (see the HCTM Application Security Standards) fully, partially, or not at all. This support overview fits naturally in the chosen approach:

1. One should leverage the Play framework functionality when it fully supports a security feature as described in the standard.
2. One should use the Play framework with custom code when the Play framework features either partially support the standard or when it eases the implementation of the standard.
3. One should write custom code when the framework does not support the standard or when the framework features are insecure.

The table below summarizes the Play framework supported security features. The columns are the different security concerns. Each row represents the standard number as represented in the HCTM Application Security Standards document. For instance, the second row represents the second standard for each concern. Each cell thus corresponds to a standard in the standards document. For instance, the third cell in the second row represents the second access control standard, namely access control must be enforced for all resources. Each cell contains a short title representing the standard, an icon representing the level of support in the Play framework for that standard, and the section in this document that covers guidance for that standard.

✓ Indicates that the standard is fully supported.

**±** Indicates that the standard is partially supported.

**⃠** Indicates that the framework (partially) supports the standard, but Cigital discourages its use.

🗶 Indicates that the standard is not supported.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Standard**  **Nr.** | **Authentication** | **Session Management** | **Access Control** | **Input Handling** | **Error Handling** | **Cryptography** | **Secure Communication** | **HTTP Security** |
| **1** | *Server-side*  **±**  4.1.1 | *128 bit ID*  **±**  4.2.1 | *Server-side*  🗶  *4.3.1* | *Server-Side*  ✓  *4.4.1* | *Separate Log*  ✓  *4.5.1* | *Server-Side*  **⃠**  4.6.1 | *TLS*  *✓*  4.7.1 | *Accept set of HTTP methods*  ✓  *4.8.1* |
| **2** | *All resources*  ✓  4.1.2 | *Session timeout*  **±**  *4.2.2* | *All resources*  🗶  *4.3.2* | *Validate at entry point and business layer*  **±**  4.4.1 | *No detailed error*  ✓  *4.5.2* | *Crypto libraries*  **⃠**  4.6.1 | *Valid Certificate*  **±**  4.7.1 | *Content Type header*  ✓ |
| **3** | *Generic Error Message*  **±**  4.1.3 | *Regenerate session*  🗶 | *No Directory Listing*  ✓  *0* | *Encode and validate all input*  **±**  *4.4.1* | *Server-Side*  ✓  *4.5.1* | *Key protected location*  **±** | *Log backend TLS failures*  🗶 | *Spoof headers*  🗶 (Needs to be done by intercepting proxy, not framework) |
| **4** | *Account lockout*  🗶  4.1.4 | *Do not disclose id*  🗶 | *Anti CSRF*  ✓  4.3.4 | *Reject + message*  **±**  *4.4.1* | *Log specific data*  🗶 | *Secure Random*  🗶 | Sensitive data in body  ✓  4.7.1 | *X-Frame-Options*  ✓  *4.8.2* |
| **5** | *Account unlock*  🗶 | *Cookies: path*  ✓  4.2.3 | *Log decisions*  ✓  4.5.1 | *Injection controls*  **±**  4.4.2 | *Analysis Tool*  🗶 | *Change algorithms*  **⃠**  4.6.1 |  | *Do not expose system info in headers*  **±** |
| **6** | *HTTPS*  ✓  4.1.5 | *Cookies: secure*  ✓  4.2.3 | *No direct access to implementation objects*  🗶 | *Output encode*  **±**  4.4.3 | *Centralized Logging*  ✓  *4.5.1* | *Protect sensitive data at server*  **±** |  |  |
| **7** | *Hash password*  **⃠**  4.1.6 | *Cookies: httponly*  ✓  4.2.3 |  | *Log failures*  ✓  4.5.1 | *Output Encode Log*  🗶 | *Protect sensitive data at client*  **±** |  |  |
| **8** | *No password at client*  🗶  4.1.7 | *Cookies: maxage and expire*  ✓  4.2.3 |  | *XML External Entities*  🗶  ✓ Play 2.3.5  4.4.4 |  | *Secure Hash*  🗶 |  |  |
| **9** | *Sufficient entropy*  🗶 | *Random id*  🗶 |  | *Implementation level types*  ✓  *4.4.5* |  | *Secure MAC*  **⃠**  4.6.1 |  |  |
| **10** | *Complex password*  🗶 |  |  |  |  | *Secure Crypto*  **⃠**  4.6.1 |  |  |
| **11** | *No Common passwords*  🗶 |  |  |  |  | *No Padding Errors*  *?* |  |  |
| **12** | *Log decisions*  ✓  4.5 |  |  |  |  | *Secure Asymmetric*  🗶 |  |  |
| **13** | *Credentials shall be encrypted*  **⃠**  0 |  |  |  |  | *Secure Signature*  🗶 |  |  |
| **14** | *Forgot password*  🗶 |  |  |  |  | *Secure Key Exchange*  🗶 |  |  |
| **15** | *Reauthentication*  **±**  4.1.9 |  |  |  |  |  |  |  |
| **16** | *No account reuse*  🗶 |  |  |  |  |  |  |  |

# Technical Guidance

This section lists the technical guidance for the standards mentioned in the HCTM Application Security Standards document.

Each element contains the following elements:

* **Title**: a brief description of the main topic for the technical guidance.
* **Technical information**: the actual technical information.

## Authentication

### Authentication controls shall be enforced on the server side

Play uses action composition to ensure that actions are authenticated. Action composition is the ability to compose multiple actions together in a chain. Each action can either process the request before delegating that request to the next action or it can dispose off the request itself. For example, a Play action can, not only modify the result but also can generate the result itself (e.g. decide to deny access to an action on a failed login ).

**±**

Play has a built in authenticator action, namely Security.Authenticator. This action should be extended to implement default authentication behavior.

import play.\*;

import play.mvc.\*;

import play.mvc.Http.\*;

public class Secured extends Security.Authenticator {

@Override

public String getUsername(Context ctx) {

//obtain and return the username

return username;

}

@Override

public Result onUnauthorized(Context ctx) {

return redirect(routes.Application.login());

}

}

Code Fragment 1: An authentication class should extend the Authenticator class.

The getUsername method obtains the username of the current authenticated user. By default, the method obtains the username from the integrity-protected session cookie.

However, for HCTM applications, Cigital recommends storing a session id in the cookie (see Section 4.2) and using that session id to obtain information related to the user from a database (such as username) or from a session cache (such as time of last request). A session cache is a Play framework cache object where the session id is used as a key.

Where scalability is critical, Cigital recommends using the default approach of the Play Framework. The information that is stored in the session should, however, be encrypted and integrity protected (see Section 4.6).

If the getUsername method returns a value, then the authenticator considers the user to be logged in, and lets the request proceed. If however the method returns null, then the authenticator will block the request, and instead invoke onUnathorized.

The unauthorized method redirects the user to a specified page. The default is a standard 401 page, but Cigital recommends redirecting the user to the login page (for usability reasons).

Cigital recommends that access control is implemented via another action or filter, as described in Section 4.3.1. The access control action must be invoked after successful authentication.

Additionally, Cigital recommends using server-side form validation as described in Section 4.4.1**Error! Reference source not found.** for any errors that a user may encounter when entering usernames and passwords.

### All protected resources must be authenticated before access

The Play framework routes all requests through the controllers (via the routes file). Authentication for all entry points (in the controllers) should be enforced via the usage of the @Security.Authenticated annotation. Using this annotation on a controller ensures that the authentication method is called before every action that is defined in the controller.

✓

@Security.Authenticated(Secured.class)

public MyController extends Controller {

//controller code here

}

Code Fragment 2: The Security.Authenticated annotation ensures that actions are authenticated.

Only requiring authentication for certain actions is strongly discouraged, as it increases the risk that actions are publically exposed; e.g. due to not revising all the annotations on code changes.

public MyController extends Controller {

//This is not recommended

**@Security.Authenticated**(Secured.class)

public static index() {

//action specific code

}

}

Code Fragment 3: The Security.Authenticated action should only be used at Controller level rather than at action level.

Requiring authentication for assets uses the same procedure. Do not use the default Assets controller in the routes file for private assets, but rather use a custom controller that handles authentication and authorization.

More Information:

*https://www.playframework.com/documentation/2.4.x/api/java/play/mvc/Security.Authenticated.html*

### Applications shall respond with a generic error message when users enter the wrong username/password combination

The framework does not support authentication out of the box. Therefore, custom code should be used to return a generic error message when authentication fails, as illustrated below. Cigital recommends leveraging the validation framework to return this error message to the user, as described in Section 4.4.1**Error! Reference source not found.**.

**±**

public String validate(String userid, String password) {

if (!User.authenticate(userid, password)) {

return "Login Failed: Invalid username or password";

}

return null;

}

Code Fragment 4: Authentication code should return a generic error message when it fails.

### Account lockout mechanisms shall be implemented

The framework does not support account lockout. Cigital recommends storing the number of failed login attempts in a database (per user) and not in the session, cache, or flash objects for the following reasons:

🗶

* Storing the number of login attempts per session enables an attacker to guess a password an unlimited number of times via different sessions.
* Play cache does not guarantee that the data in the cache is not deleted before cache expiry. Therefore, an attacker may attempt to login more than explicitly allowed.
* The flash object stores data only for one request.

### PII handled by the application shall traverse encrypted links

Cigital recommends using a different end point, such as Apache, as TLS termination link for the Play server as it eases load-balancing, scalability, and fault tolerance. However, the Play framework can be configured to serve requests over HTTPS by using the following system settings (use them as parameters to the start script):

✓

* **https.port=443** ensures that Play serves requests over HTTPS.
* **http.port=disabled** ensures that Play does not serve requests over HTTP.
* **https.keyStore=/path/to/keystore** uses the keystore that stores the SSL certificate used by Play.
* **https.keyStorePassword=changeme** stores the password used to decrypt the key store.
* **jdk.tls.ephemeralDHKeySize=2048** configures the key size.
* **jdk.tls.rejectClientInitiatedRenegotiation=true** rejects client side renegotiation.

With respect to communication with other internal and external servers, the Play framework validates (as a client) certificates according to current standards. Cigital advises against changing the default values.

The framework does not validate whether the certificate has been revoked, as that is too much overhead for scalable applications. However, if the business context requires this, it can be enabled via OCSP in JSSE or via hard-coded CRLs.

Enabling OCSP in JSSE is done as follows.

First, OSCP needs to be enabled by configuring the system properties as follows:

java -Dcom.sun.security.enableCRLDP=true -Dcom.sun.net.ssl.checkRevocation=true

Code Fragment 5: OCSP can be enabled by using command line settings.

Second, certificate revocation can be enabled in the client by setting the checkRevocation property in the application configuration file.

ws.ssl.checkRevocation = true

Code Fragment 6: certificate revocation can be enabled in the client by using the checkRevocation property.

A static CRL list can be enabled by setting the ws.ssl.revocationLists property in the configuration file.

ws.ssl.revocationLists = [ "http://example.com/crl" ]

More information:

*https://www.playframework.com/documentation/2.3.x/ConfiguringHttps*

*https://www.playframework.com/documentation/2.3.x/CertificateValidation*

*https://www.playframework.com/documentation/2.3.x/ExampleSSLConfig*

*https://www.playframework.com/documentation/2.0/HTTPServer*

### Account passwords shall be stored in protected form at the server-side

The Play framework has a Crypto API that supports encryption, signing, and hashing. We do not recommend using the *cryptoPwdHash* method of this API as it uses the insecure MD5 method. Note that newer 2.x version of the framework does not support any password hashing functionality.

**⃠**

//This is not recommended

import play.libs.Crypto;

public class PasswordHandler{

public String protect(String pwd) {

**cryptoPwdHash**(pwd);

}

}

Code Fragment 7: Do not use the cryptoPwdHash function in the 1.x version of the framework.

Cigital recommends implementing the crypto-standards using any JCE provider.

### Account passwords shall never be stored at the client-side in insecure storage

🗶

Cigital advises against storing credentials or other sensitive information in session or flash objects, as their data is stored unencrypted in cookies at the client-side. An attacker may thus read the contents of these cookies to obtain passwords.

//This is not recommended

String password = …

**session**(“password”, password);

**flash**(“password”, password);

Code Fragment 8: Do not store credentials in the session or flash objects.

If sensitive information needs to be stored in the session or flash objects, then Cigital recommends using standard Java libraries to encrypt and integrity-protect that information.

### All authentication credentials for accessing services external to the application shall be encrypted

**⃠**

The Play framework supports encryption with AES. However, Cigital suggests using standard Java libraries as the framework’s default AES parameters (ECB mode) are insecure.

//This is not recommended

import play.libs.Crypto;

public class FancyOperation{

public String protect(String sensitive) {

return **encryptAes**(sensitive);

}

public String unprotect(String protected) {

return **decryptAes**(sensitive);

}

}

Code Fragment 9: Do not use the encryptAes functionality.

### Users shall be required to re-authenticate to the web application before executing any sensitive transaction

**±**

Cigital suggests using the same mechanism as described for regular authentication (Section 4.1.1). Additionally, Cigital suggests creating a custom annotation that invokes the authenticator class and annotating the sensitive actions with this annotation.

An example of a custom class invoking the authenticator class is as follows:

public class SensitiveActionAuthenticator extends Action.Simple {

public Result call(Http.Context ctx) throws Throwable {

//verify authentication

doAuthentication();

//delegate to the real action

return delegate.call(ctx);

}

}

Code Fragment 10: Create an authenticator for sensitive actions via action composition.

Annotate sensitive actions with the class name.

@With(SensitiveActionAuthenticator.class)

public static Result sensitiveaction() {

return ok("It works!");

}

Code Fragment 11: Annotate sensitive actions with a custom authenticator.

More Information:

*https://www.playframework.com/documentation/2.4.x/JavaActionsComposition*

## Session Management

### The session IDs shall be at least 128 bits and random

The Play Framework (2.0 and up) does not implement session ids, as it focuses on client sessionless tokens for performance. However, for HCTM projects Cigital recommends generating a cryptographically secure random number and storing that number in the session object. A random number can be generated with the SecureRandom class.

🗶

// Generate a unique id

String session\_id = session("session\_id");

if(session\_id ==null) {

//generate a random number with SecureRandom.

session\_id = generateRandomNumber();

session("session\_id", session\_id);

}

Code Fragment 12: Generate session ids with a secure random number generator.

More Information:

[*http://www.cigital.com/justice-league-blog/2009/08/14/proper-use-of-javas-securerandom/*](http://www.cigital.com/justice-league-blog/2009/08/14/proper-use-of-javas-securerandom/)

### Sessions shall be invalidated

The Play framework does not implement session invalidation at server side. Custom code can achieve this as follows:

🗶

* Store the session information in a session cache and use cache invalidation to invalidate sessions after a fixed period, or
* Store the time of the last request in the session object and use action composition to update the time and invalidate the session when expired.

Sessions can be invalidated via a Play session cache as follows.

String session\_id = … //obtain session id of the current user

String time = … //obtain the time of the last request

cache.set(session\_id + “last\_request\_time”,time, 60 \* 15);

Code Fragment 13: Session cache can be used for invalidation.

The example code stores the last\_request\_time in the session cache of a user (represented by session\_id) and invalidates it after 15 minutes. The first argument of the cache.set function is the item key. To store information per session, one can append the session\_id to a key such as last\_request\_time.

The second argument is the value that is stored in the session; e.g. “*Sunday October 19th*”.

The last argument states that this value must be invalidated after 15 minutes, i.e. 60 times 15 seconds.

Note that the Play framework may remove data from the cache before the time period is over. For instance, in the above example the Play framework invalidates the data after at latest 15 minutes.

A session can also be invalidated via action composition. Add the following code to the authentication action.

// verify whether the session is expired

String previousTick = session("userTime");

if (previousTick != null && !previousTick.equals("")) {

long previousT = Long.valueOf(previousTick);

    long currentT = new Date().getTime();

    long timeout = Long.valueOf(Play.application().configuration().getString("sessionTimeout")) \* 1000 \* 60;

if ((currentT - previousT) > timeout) {

     // session expired

        session().clear();

        return null;

    }

}

// update time in session

String tickString = Long.toString(new Date().getTime());

session("userTime", tickString);

Code Fragment 14: Sessions can be invalidated by storing the last request time in the cookie.

More Information:

*https://www.playframework.com/documentation/2.4.x/JavaCache*

*https://www.playframework.com/documentation/2.4.x/ScalaSessionFlash*

*http://www.poornerd.com/2014/04/01/how-to-implement-a-session-timeout-in-play-framework-2/*

Note that the session id generation of the last URL is not very secure. In Java, a long is only 64 bit, while a 128bit string is preferred for session ids.

### Cookies attributes

The value of various cookie attributes can be configured via the application.conf file.

✓

Applications using the Play framework greater than 2.0 should use the following options:

session.secure=true

session.httpOnly=truesession.maxAge=3600

Code Fragment 15: Various cookie attributes can be set via the application configuration file.

Applications using older versions of the framework should use the following options:

application.session.secure=true

application.session.httpOnly=trueapplication.session.maxAge=3600

Code Fragment 16: Various cookie attributes can be set via the application configuration file.

Other cookie attributes such as *expires* and C*ache Control* can be configured per action or for all actions.

The setHeader method sets cookie attributes for individual actions by adding the header to the response. Its first parameter is the attribute, while its second parameter is the value.

public static Result index() {

response().setHeader(“Cache Control”, "No-store");

return ok("<h1>Hello World!</h1>");

}

Code Fragment 17: Cookie attributes can be configured per action.

Composed actions or filters set cookie attributes for all actions by intercepting the responses and adding the cookies. It is hard to do using Java. As projects can contain both Java and Scala code, Cigital recommends using Scala code to achieve this.

More Information: [*https://github.com/playframework/playframework/blob/master/framework/src/play-filters-helpers/src/main/scala/play/filters/headers/SecurityHeadersFilter.scala*](https://github.com/playframework/playframework/blob/master/framework/src/play-filters-helpers/src/main/scala/play/filters/headers/SecurityHeadersFilter.scala)

*http://www.playframework.com/documentation/2.0/ScalaSessionFlash*

## Access Control

### Access Control shall be enforced server-side

The Play framework does not implement access control. However, Cigital recommends using the deadbolt plug-in.

🗶

More information:

*http://deadbolt.ws/#/home*

### Access control shall be enforced for all resources

🗶

The deadbolt plug-in restricts access to a resource to a specific group of users. This group is typically defined by the roles held by subjects. Each protected resource can define its own requirements for access, or delegate access decisions to a central component that handles authorization concerns.

The deadbolt plug-in provides two mechanisms to declare these access restrictions, namely at the template level (view) and another at the controller level. The general idea is that the Play framework routes all requests through the deadbolt plug-in view the controllers (via the routes file). The deadbolt plug-in then enforces authorization for all entry points (in the controllers).

Cigital recommends enforcing access control at the controller level rather than at the template level. The plug-in supports various annotations, of which @SubjectPresent and @Restrict are the most important ones.

Annotating a controller (or action) with @SubjectPresent ensures that only logged in users can execute the actions in that controller (or that specific action).

@SubjectPresent

public MyController extends Controller {

//controller code here

}

Code Fragment 18: Access control is enforced for all resources via the deadbolt plugin.

Annotating a controller with @Restrict allows users with the necessary role(s) to call the actions of that controller, while blocking all other users. Roles are defined in one or more @Group annotations. Within a group, roles are conjunctive (AND), and between groups the role groups are disjunctive (OR).

For instance, @Restrict({@Group("admin"), @Group("developer", "techlead")}) means that the user must either have the admin role or the developer and techlead roles.

Even though deadbolt supports exclusion of access to groups (expressed via !), Cigital recommends listing the roles that can access certain functionality explicitly.

//This is not recommended

@Restrict({@Group(!“user”)})

Public MyController extends Controller {

//controller code here

}

Code Fragment 19: Access control supports restrict from.

@Restrict({@Group(“admin”)})

Public MyController extends Controller {

//controller code here

}

Code Fragment 20: Annotations should explicitly mention the groups that have access to the actions.

Cigital encourages using access control at a controller level, rather than at an action level, as it decreases the risk that actions are publically exposed; e.g. due to not revising all the annotations on code changes.

//This is not recommended

Public MyController extends Controller {

//controller code here

@SubjectPresent

public static Result myAction() {

…

}

}

Code Fragment 21: Access control can be enforced per method.

Cigital recommends refining controller annotations for actions when necessary.

@SubjectPresent

Public MyController extends Controller {

//controller code here

@Restrict({@Group{“admin”}})

public static Result myAction() {

…

}

}

Code Fragment 22: Restrictions can be refined.

### Directory browsing shall be disabled

The Play framework disables directory listing by default, as all requests are redirected via the routes file. Cigital discourages implementing a directory listing action.

✓

### The application shall generate strong random anti-CSRF tokens

✓

The Play framework supports anti-CSRF protection mechanisms. The code should use global filters and form tokens.

A global CSRFFilter is created as follows.

public class Global extends GlobalSettings {

@Override

public <T extends EssentialFilter> Class<T>[] filters() {

return new Class[]{CSRFFilter.class};

}

}

Code Fragment 23: A global CSRF filter is created in the GlobalSettings class.

In your form template, annotate the form with a CSRF form field as follows.

@form(routes.myController.post()) {

@CSRF.formField

...

}

Code Fragment 24: A CSRF field should be added to forms.

This may render the following form:

<form method="POST" action="/items">

<input type="hidden" name="csrfToken" value="1234567890abcdef"/>

...

</form>

Code Fragment 25: Result of rendering a CSRF form field.

Cigital discourages the use of per action checks, as it is error prone. Indeed, a developer may accidentally forget to annotate an action with CSRF-protection checks. Therefore, enabling global CSRF-protection (as illustrated above) is preferred.

//This is not recommended

// Actions that are session authenticated are annotated with @RequireCSRFCheck

@RequireCSRFCheck

public static Result save() {

// Handle body

return ok();

}

//Actions that render forms are annotated with @AddCSRFToken

@AddCSRFToken

public static Result get() {

return ok(form.render());

}

Code Fragment 26: All actions that are session authenticated should be annotated with RequireCSRFCheck.

Additionally, Cigital discourages adding the token to the action URL.

//This is not recommended

@import helper.\_

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

...

}

Code Fragment 27: CSRF tokens should not be added to the action URL.

More information:

*https://www.playframework.com/documentation/2.4.x/JavaCsrf*

## Input Validation

### All input validation shall be performed on the server side

The Play framework provides a forms API for handling, rendering, decoding, and validation of forms. The easiest way to handle form submission is to define a form (play.data.Form) that wraps an existing data class (e.g. Login.class with username and password fields or a Profile.class with name and contact details).

✓

Form<Login> loginForm = Form.form(Login.class)

Code Fragment 28: A form can wrap an existing data class.

Another possibility is using a DynamicForm to retrieve data from an HTML form that is not related to a model class. Cigital discourages its use, as it decentralizes validation code.

Forms are validated at the server side by routing the form input during submission to a class that performs semantic checks on the form data. All HTML forms must be annotated with @helper.form to tell the Play framework which route (e.g. routes.application.authenticate or controllers.Application.updateProfile) forms should be submitted to. The annotation @helper.inputText creates a form field with the given name, size, and id.

@helper.form(action = routes.Application.authenticate()) {

<form>

<input…>

</form>

}

Code Fragment 29: forms should be redirected to controller actions.

@helper.form(routes.Application.updateProfile) {

@helper.inputText(myForm(“email”), size -> 30)

@helper.inputText(myForm(“birthdate”), size->12)

}

Code Fragment 30: forms should be redirected to controller actions.

The routes file (conf/routes) should be configured to contain the page (e.g. /login or /updateProfile) and the controller action (e.g. controllers.Application.authenticate() or controllers.Application.updateProfile()) that will process and validate the data submitted by the form.

POST /login controllers.Application.authenticate()

Code Fragment 31: forms should be redirected to controller actions.

POST /login controllers.Application.updateProfile()

Code Fragment 32: forms should be redirected to controller actions.

Naturally, the controller action (e.g., authenticate) also needs to validate the data submitted via the form. The simplest way to achieve this is by implementing a validate() method in the data class (e.g. Login.class or Profile.class) .

public String validate() {

if (!User.authenticate(userid, password)) {

return "Login Failed: Invalid username or password";

}

return null;

}

Code Fragment 33: the controller actions should validate the input.

Notice that this code delegated the validation to the User class of the model of the application. If validation fails a String with the error message is returned, otherwise, no error message is returned (null). Note that data is validated at entry points (the form), but also at the business layer (the user). Moreover, the business validation code resides in the business layer. Validation code may be centralized via annotations and Play’s validation framework. Example annotations include @Required, @Email, @MinLength, @MaxLength, and @Pattern. The @Required annotation ensures that the annotated field is submitted. The @Email annotation verifies whether the annotated field is an e-mail address. The @MinLength annotation verifies whether the annotated field has the given minimum length. The @MaxLength annotation verifies whether the annotated field has the given maximum length.The @Pattern annotation verifies whether the annotated field matches the given regular expression.

public class UpdateProfile {

@Required

@Email

public String email;

}

The validation then uses the hasErrors method of the Form object to handle binding errors in the controller action.

public static Result authenticate() {

Form<Login> loginForm = Form.form(Login.class).bindFromRequest();

if (loginForm.hasErrors()) {

return badRequest(login.render(loginForm));

} else {

//execute business code related to form

}

}

Code Fragment 34: the hasErrors method should be used to identify whether the form had any errors associated with it.

This code introduces a number of new concepts. First, if validation fails, it returns a 400 Bad Request status (badRequest(render)), rendering the login page with the form that had the failed validation. By passing this form back, the code can extract any validation errors from the form and render them back to the user.

If the validation was successful, then the code should execute the action associated with the form.

The template needs to display the error message when validation fails. This can be done by passing the invalid form back to the template via the @if(form.hasGlobalErrors) and the @form.globalError.message directives.

@if(form.hasGlobalErrors) {

<p class="error">

@form.globalError.message

</p>

}

Code Fragment 35: the hasGlobalErrors template should be used to display global errors.

The updateProfile example requires more complex code. The template should be able to render error messages per input field, rather than per form. This can be done by passing the invalid field back to the template via the following construct.

@for(error <- form(“email”).errors) {

<p>@error.message</p>

}

Code Fragment 36: the form(“fieldname”).errors construct should be used to display field errors.

Additionally, form field strings, such as a birth date, should be mapped to data objects, such as LocalDate, and vice versa. The Play framework provides the formatter object to achieve this. This

Formatters.register(LocalDate.class, new SimpleFormatter<LocalDate>() {

@Override

public LocalDate parse(String input, Locale l) throws ParseException {

return LocalDate.parse(input);

}

@Override

public String print(LocalDate localDate, Locale l) {

return localDate.toString();

}

});

Code Fragment 37: Formatter objects convert data objects to form field strings and vice versa.

The Play framework generates an array of errors, when the binding fails. The array contains the following:

["error.invalid.<fieldName>", "error.invalid.<type>", "error.invalid"]

Code Fragment 38: The Play Framework generates an array of errors when binding fails.

These errors can be displayed in a similar way as defined before.

More information:

[*https://www.playframework.com/documentation/2.4.x/JavaForms*](https://www.playframework.com/documentation/2.4.x/JavaForms)

### The application shall use controls to avoid injection vulnerabilities

**±**

Applications may use the Ebean component of the Play framework to store and retrieve data. Cigital recommends using the Ebean component rather than creating SQL statements. Within the Ebean component, Cigital discourages the use of raw SQL statements.

To enable Ebean, first add javaEbean to the project’s dependencies:

libraryDependencies += javaEbean

Next, define a default Ebean server by adding the following line to the application configuration file conf/application.conf:

ebean.default="models.\*"

Creating separate databases for development and testing is done by naming the Ebean servers via the ebeanconfig.datasource.default property. The servers can then be accessed by name in the code.

EbeanServer testServer = Ebean.getServer(“testServer”);

EbeanServer defaultServer = Ebean.getServer(null);

Code Fragment 39: The Ebean getServer method obtains the server with given name.

Ebean supports fetch, save, and delete queries.

The find method fetches an object that matches given criteria. The following example obtains a customer with id 4.

// find a customer by their id

Customer customer = Ebean.find(Customer.class, 4);

Code Fragment 40: The Ebean find method finds an object that matches given criteria, in this call a customer with id 4.

The findList method fetches multiple objects that match given criteria. The following example fetches a list of customers whose name starts with ‘Rob’. Notice that the general SQL language constructs, such as where, like, and orderBy, can be used.

// find a list...

List<Customer> customers =

Ebean.find(Customer.class)

.where().like(“name”, “Rob%”)

.orderBy(“name desc”)

.findList();

Code Fragment 41: The Ebean findList method finds multiple objects; in this case all customers whose name start with 'Rob'.

The save method persists changes to an object. The following example first changes the status and the shipping date of an order and then saves the changes.

Order order = Ebean.find(Order.class, 12);

order.setStatus(OrderStatus.SHIPPED);

order.setShipDate(...);

// this will save the order

Ebean.save(order);

Code Fragment 42: The Ebean save method saves changes to objects.

The delete method removes the object from the database. The following example deletes a task with ID 34L.

// Delete a task by ID

Task.find.ref(34L).delete();

Code Fragment 43: The Ebean delete method removes the given objects from the database.

More complex examples, including transactions and caching, are available in Ebean’s user guide.

Cigital recommends using Ebean for all queries. However, Cigital discourages the use of raw SQL statements within Ebean, because it could render the system insecure, despite the use of eBean’s ORM.

String sql

= " select order\_id, o.status, c.id, c.name,

sum(d.order\_qty\*d.unit\_price) as totalAmount"

+ " from o\_order o"

+ " join o\_customer c on c.id = o.kcustomer\_id "

+ " join o\_order\_detail d on d.order\_id = o.id "

+ " group by order\_id, o.status ";

**RawSql rawSql =**

RawSqlBuilder

.parse(sql)

// map result columns to bean properties

.columnMapping("order\_id", "order.id")

.columnMapping("o.status", "order.status")

.columnMapping("c.id", "order.customer.id")

.columnMapping("c.name", "order.customer.name")

.create();

Query<OrderAggregate> query = Ebean.find(OrderAggregate.class);

query.setRawSql(rawSql)

Code Fragment 44: The usage of Raw SQL statements within the Ebean framework is discouraged.

Additionally, Cigital discourages the use of SQL statements, as illustrated below.

//This is not recommended.

String sql = "...";

Connection conn = play.db.DB.getConnection();

try {

Statement stmt = conn.createStatement();

try {

stmt.execute(sql)

} finally {

stmt.close();

}

} finally {

conn.close();

}

Code Fragment 45: The usage of SQL statements is discouraged.

More Information:

[*https://www.playframework.com/documentation/2.3.0/JavaEbean*](https://www.playframework.com/documentation/2.3.0/JavaEbean)

*http://www.avaje.org/doc/ebean-userguide.pdf*

### All untrusted data that is output to the user shall be properly escaped/encoded

**±**

The Play framework encodes data that can be represented in the templates; i.e. HTML, CSS, and JavaScript. However, the framework does not factor in contextual encoding approaches. Therefore, the application should use output encoding libraries to output encoded data.

Cigital recommends creating a global filter for output encoding. That global filter should call an output encoding library such as OWASP Java Encoder. Calling the library as a global filter ensures that the developers do not need to remember to call the library, but that the framework invokes it on every request.

### Input handling libraries shall be configured to use secure-defaults

The XML processor of the Play framework (2.3.4. and lower) is vulnerable to External Entity attacks. Therefore, Cigital suggests to not use the built in XML processor, but rather use an external library and configure it securely.

**±**

More Information:

*https://www.playframework.com/security/vulnerability/CVE-2014-3630-XmlExternalEntity*

### The application shall use implementation-language-level types

The Play framework uses implementation-language-level types in the routes file. An error is thrown when the input string cannot be parsed to that type.

✓

## Error Handling and Logging

Most error handling has been described in Section 4.1.1.

### The application shall maintain a separate security log

✓

This section focuses on how to use the Play framework to create security logs. The HCTM Application Security Standards document elaborates on what to log.

The application can use the Logging component of the Play framework to log various decisions.

Cigital recommends creating a separate security logger and use that logger in the method that requires logging.

A new logger can be created via the Logger.of factory method with a name argument:

final Logger.ALogger securityLogger = Logger.of("security.authentication");

Code Fragment 46: a separate security logger should be created.

In the method requiring logging, a user can invoke the logger as follows:

securityLogger.info(…);

Note that each of the user controllable parameters should be output encoded to avoid log injection.

Cigital discourages using the default logger for both application events as well as security events, as it increases the chances that an attacker can pollute the security logs.

//This is not recommended

final Logger.ALogger logger = Logger.of("application");

//This is not recommended, as it writes to the default logger

Logger.debug("Attempting risky calculation.");

Code Fragment 47: Cigital discourages using the default logger for security events.

Cigital recommends creating a Logback configuration file for custom secure configuration. This configuration file should be stored under conf/logger.xml, while the configuration in application.conf should be removed.

An example LogBack configuration file is as follows:

<configuration>

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>/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 %n%message%n%xException%n</pattern>

</encoder>

</appender>

<appender name="ACCESS\_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>/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

Logs should not be stored on the application server, but rather logged to a central location with appropriate controls.

More information:

*https://www.playframework.com/documentation/2.3.x/JavaLogging*

[*https://www.playframework.com/documentation/2.3.x/SettingsLogger*](https://www.playframework.com/documentation/2.3.x/SettingsLogger)

[*http://logback.qos.ch/manual/appenders.html*](http://logback.qos.ch/manual/appenders.html)

### The application error pages shall not display sensitive information

The Play framework handles all exceptions thrown by the code and redirects the user to an error page. This page contains the stacktrace during development. Cigital suggests running the Play framework in production mode or creating custom error pages.

✓

When an exception occurs in a Play application, the onError operation will be called. The default is to use the internal framework error page. This can be overridden as follows:

public class Global extends GlobalSettings {

public Promise<Result> onError(RequestHeader request, Throwable t) {

return Promise.<Result>pure(internalServerError(

views.html.errorPage.render(t)

));

}

}

Code Fragment 48: Cigital recommends using a custom error page that displays a simple error message.

The errorPage is a template that renders the error information. That template should not print the stacktrace of its parameter of type Throwable.

Additionally, the Play framework discloses sensitive information when actions are not found. If the framework does not find an action method for a request, the onHandlerNotFoundoperation is called:

public class Global extends GlobalSettings {

public Promise<Result> onHandlerNotFound(RequestHeader request) {

return Promise.<Result>pure(notFound(

views.html.notFoundPage.render(request.uri())

));

}

}

Code Fragment 49: Cigital recommends using a custom error page that displays a simple error message.

The notFoundPage is the template that renders the error information.

Moreover, the Play framework discloses internal information when the parameter types were incorrect. If a route was found, but it was not possible to bind the request parameters, the onBadRequest operation is called.

public class Global extends GlobalSettings {

public Promise<Result> onBadRequest(RequestHeader request, String error) {

return Promise.<Result>pure(badRequest("Do not hack the URI!"));

}

}

Code Fragment 50: Cigital recommends using a custom error page.

More Information:

*https://www.playframework.com/documentation/2.3.x/JavaGlobal*

## Cryptography and Data Protection

### All cryptographic functions used to protect server-side application secrets shall be implemented server side, preferably by a central component

**⃠**

Cigital discourages using the Play Crypto API, as it does not use secure defaults nor does it implement all the cryptographic algorithms to store data securely. More details on how to store data in a secure way is discussed in the Spring Framework document (Security Standards and Guidelines – Spring Framework).

## Secure Communication

### TLS shall be used for all connections

Cigital recommends configuring HTTPS using the same settings as described in Section 4.1.5.

✓

## HTTP Security

### The application shall only accept a defined set of HTTP request methods

✓

The Play framework only accepts the HEAD, GET, POST, PUT, or DELETE HTTP operations. The accepted operations for each of the controller actions are explicitly defined in the routes file. For instance, the following routes entry only accepts the GET HTTP operation.

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

Code Fragment 51: The routes file defines the methods the application accepts.

Cigital recommends using REST semantics for each of the route entries; i.e. GET for reading operations, DELETE for delete operations, etc.

### Every HTTP response shall contain a content type header specifying a safe character set

✓

The Play framework automatically sets the content type header for text-based responses, i.e. text/xxx and utf-8 (where xxx can be text/plain, text/html, etc.

For other types of responses, the content type and its encoding can be specified during the generation of the Result in the controller, as follows:

public static Result index() {

response().setContentType("text/html; charset=iso-8859-1");

return ok("<h1>Hello World!</h1>", "iso-8859-1");

}

Code Fragment 52: The content type of a response is set via the setContentType method.

More information:

*https://www.playframework.com/documentation/2.4.x/JavaResponse*

### The HTTP header, ‘*X-Frame-Options*’ and X-Content-Type-Options shall be used

The HTTP header X-Frame-Options should be enabled by adding the SecurityHeadersFilter class to the list of filters in the Global object:

✓

public class Global extends GlobalSettings {

public <T extends EssentialFilter> Class<T>[] filters() {

return new Class[]{SecurityHeadersFilter.class};

}

}

Code Fragment 53: The X-Frame-Options header should be enabled via the SecurityHeadersFilter.

The X-Content-Type-Options header shall be set to nosniff in the in the application.conf file.

X-Content-Type-Options=nosniff

Code Fragment 54: X-Content-Type-Options is set to nosniff in the application.conf file.

More information:

*https://www.playframework.com/documentation/2.4.x/SecurityHeaders*

### Headers shall not contain sensitive information

The Play framework does not disclose sensitive information in its headers. If sensitive headers are added by plugins, the recommended way to remove them is via the Global object, as described in Section 4.2.3.

✓

1. https://www.playframework.com/documentation/2.4.x/Home and https://www.playframework.com/documentation/2.3.x/Home [↑](#footnote-ref-1)
2. https://github.com/playframework/playframework/tree/master/ [↑](#footnote-ref-2)
3. Bosch - HCTM Application Security Standards, Version 3.01(Web application security standards) [↑](#footnote-ref-3)