Европейски парламент Parlamento Europeo Evropský parlament Europa-Parlamentet Europäisches Parlament
Euroopa Parlament Ευρωπαϊκό Κοινοβούλιο European Parliament Parlement européen Parlaimint na hEorpa
Europski parlament Parlamento europeo Eiropas Parlaments Europos Parlamentas Európai Parlament
Parlament Ewropew Europees Parlement Parlament Europejski Parlamento Europeu Parlamentul European
Európsky parlament Evropski parlament Euroopan parlamentti Europaparlamentet

EP DEVELOPMENT STANDARDS OVERVIEW

European Parliament

Version v1.1, 2017-07-21

Table of Contents

The current document summarizes standards for software developments made for the European Parliament.

1. EP DEVELOPMENT STANDARDS OVERVIEW

1.1. GENERAL INTRODUCTION

1.1.1. Purpose

The current document summarizes standards that apply for software developments made for the European Parliament.

1.1.2. Disclaimer

The current document specifies standards which MUST be taken into account for any new software developments produced for the European Parliament, whether it is carried out within the European Parliament premises or externalized, whether it is carried out with a fixed-price contract, or not.

All technologies and versions of libraries specified in this document are valid at the time this document is released, and until a newer version of this document is specified and published.

1.1.3. Versions compatibility

Indicative versions compatibility for frameworks, libraries or specifications (when not specified) can be found in appendix.

1.2. NORMS, STANDARDS AND TOOLING

1.2.1. Global considerations

European Parliament developments mostly rely on open source software and open standards. Any proprietary software or proprietary binary format should be avoided.

It is supported to use a layered architecture, clearly decoupling presentation layer from business layer, and persistence layer.

Integration and remoting should be kept out of the business code itself, through the use of POJOs, delegates, dependency injection, and possibly AOP.

Applications should use as much as possible a stateless conversational model, which means that it is preferable to use frameworks or solutions which save the state whether on the client side,or in a data store, as opposed to saving valuable state in sessions.

1.2.2. Development Target

Development of applications for the European Parliament should globally conform to Java standards:

- Java SE: Java Standard Edition
- Java EE: Java Enterprise Edition

A deliverable must therefore be compatible with these platforms (details of compatibility and precise supported technologies are summarized along this document), and more importantly

must NOT rely on other equivalent platforms (.Net, LAMP…etc).

1.2.2.1. Target JRE/JDK

To sum it up:

Target Java Runtime Environment (JRE) is JRE 1.8.x and above.

1.2.2.2. Target JavaEE specifications

Deliverables should be compatible with the JavaEE 6.0 specification, though **NOT as a whole**.

European Parliament does not explicitely support compatibility with a full JavaEE 6 profile (such as Web profile). Always check the supported runtime (e.g. Tomcat) and specific standards for European Parliament, as explained along this documentation.

Java Enterprise Edition is made of many specifications.

Details of clearly supported specifications are spread implicitly along this document.

Target of standards being Servlet Containers (e.g. Tomcat), not all Java EE specifications can be leveraged.

Here are the specifications newly developed applications should **NOT** directly rely on at this point in time:

- All versions from older JavaEE (J2EE) specifications that are not found in JavaEE 6.0 anymore
- Java EE Deployment
- J2EE Management
- Java Metadata Specification
- EJB (any version)
- CDI
- JACC
- JAXR
- JAX-RPC
- JAX-WS
- JCA (any version)
- JSF (any version)

Previous versions of these specifications should also obviously be avoided (e.g. EJB2.1). In case of doubt (if a specification is not mentioned in this document), projects are **NOT** allowed to presume a specification is authorized. Ask the European Parliament for confirmation.

1.2.3. Deliverables

Deliverables should respect usual best practices for code quality standards, and more as specified in specific requirements of precise projects.

Conceptually, a deliverable should contain to the very least source code, tests exercising that source code, binaries, and relevant documentation.

Here are the main constraints applied to deliverables:

- Deliverables should contain **fully documented sources** (documentation in English, e.g. javadoc in English for java code)
- It should be possible to **import and maintain deliverables in the standard European Parliament IDE and Software factory** (see "Tooling" section further in this document), regardless of the tooling used if the deliverable is developed outside of European Parliament premises.
- **Build of deliverables should be reproducible in an isolated fashion**. This means that deliverables should be accompanied with any external item needed, and with the tool used for the build. Ideally, deliverables should be easy to build in a continuous integration system.
- **Build of deliverables should be fully documented**. The level of documentation depends on the chosen build tool (see "Tooling" section further in this document).
- **Configuration** of interactions of the deliverable with other European Parliament' systems (when deployed in the internal infrastructure) **should be externalized** (see configuration standards) from the deliverables. Configuration of communication/integration interfaces (ports, URLs…etc) should also be externalized.
- **Tests should be placed in a separate folder**, or even in a separate project if needed (e.g. for integration tests or user acceptance tests).
- **Use of (Test) Doubles** (Stubs, Mocks, Dummies or Fakes) **should be clearly documented**, as well as the way to remove/replace them with real implementations when deployed in the European Parliament (this is mostly true for externally developed deliverables, though is also valuable for internal developments).

1.2.4. Tooling

1.2.4.1. IDE

The European Parliament has validated Eclipse IDE as its main tool for Java related developments. Eclipse's latest stable version is the reference.

It can be augmented with additional plugins and features such as (this is just indicative):

- SpringSource Tools Suite and required dependencies (e.g. Eclipse WTP, AJDT, Mylyn, M2Eclipse…)
- Apache LdapStudio

1.2.4.2. Build tools

The standard tool for the build of Java based deliverables in the European Parliament is Apache Maven 3.x (and above).

NodeJS/NPM (recent stable versions recommended) can also be used as a tool for building client-side web applications (Javascript/HTML/CSS development tooling).

NodeJS will NOT be used as a runtime platform in European Parliament! Only build-time usage is authorized!

1.2.4.3. Servlet container

The open source JavaEE-compatible servlet container which may be used for European Parliament developments and deployments is:

- Apache Tomcat 8.0.x.

Developments should never be done using the specifics of this deployment platforms, and should always conform to the Java EE specifications limitations mentioned upper in this document, and to other standards expressed throughout this document.

Server-specific additions to the configuration should only be added when absolutely needed.

1.2.4.4. Messaging broker

The open source JMS compatible broker to use for European Parliament developments and deployments is:

- Active MQ Artemis (formerly known as HornetQ)

Developments should never be done using the specifics of this deployment platform, and should always conform to the Java EE specifications limitations mentioned upper in this document, and to other standards expressed throughout this document. Therefore, one should only configure destinations, possibly secure them and configure dead letter queues.

It should be kept in mind that a switch to any other Messaging Broker implementation should be possible with as little work as possible (replacing the client libraries should be enough from the consumer/producer point of view).

See JMS standards later in this document for more details on usage.

1.2.4.5. Source Control Management

Inside of the European Parliament the use of an internal SVN or Git repository is supported.

External SVN repositories can be made available through https for projects where teams are split inside and outside of the European Parliament, or completely outside of the European Parliament.

1.3. PRESENTATION LAYER

1.3.1. General considerations

Projects should apply the MVC2 Pattern and use the supported frameworks described below.

1.3.2. Frameworks

1.3.2.1. Spring MVC

Spring MVC is the standard framework for creating the server side MVC implementation of web sites. It is flexible enough to render any kind of output:

- HTML pages
- PDF documents
- JSON messages
- reports
- or any kind of text generated through a template engine (Velocity, FreeMarker) …

Moreover it is easy to integrate with other libraries (like Bean Validation, "JavaScript libraries",etc.) and keeps the code easy to test.

Under the hood, Spring MVC has specialized objects which complete the main concerns of web frameworks and are easily extendable:

- Models : Containing data
- Controllers: Handling user interactions
- Views: Displaying information to end users.

Server-side page templating should be done using the Spring-powered Thymeleaf framework.

1.3.2.2. Ext-JS

Ext-JS allows elaborating a rich UI through JavaScript code.

Ext-JS is supported for teams with a good knowledge of JavaScript and should be used in conjunction with Spring MVC.

It supports all standard browsers available in the European Parliament.

Compared to JQuery or Angular, Ext-JS is more heavyweight and offers more high-level graphical components.

1.3.2.3. Angular

Angular is an intermediary alternative to Ext-JS and JQuery.

Angular is supported for teams with a good knowledge of JavaScript and should be used in conjunction with Spring MVC.

It supports all standard browsers available in the European Parliament.

Compared to JQuery, Angular offers more high-level services to manage the state of the page, and the synchronisation with the server.

Angular applications should be coded in Javascript, though Typescript can be used as long as the development team has ensured adequate tooling is available along the development toolchain.

1.3.2.4. JQuery

JQuery is a lightweight framework that makes writing JavaScript easier, more effective and greatly improves user interface. It is based on a rich DOM selector API.

For a richer UI based on JavaScript widgets, Ext-JS usage should be favoured. For a more manageable interaction with the server, while maintaining a simple UI, use in conjunction with Angular can be considered.

1.4. BUSINESS LAYER AND INTERACTION BETWEEN LAYERS

1.4.1. POJO

Use of Plain Old Java Objects (POJOs) for implementing applications' business layer is the supported coding standard. This standard is to be respected, whichever framework is used for software development.

Note that POJOs are not necessarily JavaBeans. Indeed, JavaBeans are in fact POJOs with some specific restrictions defined by the Java Beans specification.

Coding POJOs has the main advantage of keeping the framework-specific code out of your

business code.

Moreover, when using POJOs, only very basic patterns are needed. Most patterns (like factories,singletons…etc) are needed for the various frameworks that will use, expose or persist your business POJOs. This of course does not mean that these patterns are never needed, but they are usually not necessary to implement the business.

Although current trends in frameworks tend to add framework specific Java 5.0+ annotations in the code, this is not a recommended practice (at least in the European Parliament). You should only add to your code annotations that are abstract enough and business related.

You'll end up coding 2 main types of POJOs for the business layer:

- **Entities**: Entities are the parts of the system that should vary less.
- **Processes**: Processes should vary more with time and often match business activities. Usually processes are coded as **Service** (or **Manager**) Objects.

The rest of the code is usually for Data Access Objects (or preferably Repositories), remoting (synchronous or asynchronous), low level security, presentation, transformation … These are important things, yet technical things that will vary depending on the type of client, the security policy, the environment where the application will be deployed …

These technical concerns can also be coded as POJOs, but should not be referenced by your Business POJOs in order to separate technical concerns from business concerns.

1.4.2. Validation

By Validation we mean the operation of checking if object's properties respect some defined sets of rules.

Several sets of validation rules can coexist and their pertinence is context-specific but we can have a minimum set of rules that should always be respected.

Bean Validator allows usage of different sets of rules and can very easily be used to check local constraints.

European Parliament supports using a JSR 303 (Bean validation) implementation: Hibernate Validator.

With or without this framework, validation of data coming from user input, or coming from external source (web service, database…) is mandatory and will ensure better security of the application.

1.4.3. Coding to interfaces

Use of interfaces to describe and specify the interaction between different layers is mandatory. This eases the creation of mocks or stubs of functionalities for tests.

1.4.4. Dependency injection

Directly linked to the previous topics, the concept of dependency injection, also known as "Inversion of control", allows the business layer to stay clear of any presentation, persistence orintegration concerns.

Relations with other layers are defined through injection, whether through constructors, setters,or any other mechanism that would allow the business related objects to stay clear from framework-specific dependency.

Use of the Spring framework is standard in this prospect, through the use of Java configuration

and annotations, or of XML configuration if seen fit.

1.5. PERSISTENCE LAYER

1.5.1. High level persistence framework

In order to ease the implementation and maintenance of the persistence layer, applications should be coded using the **Spring Data** framework. It removes a lot of the boilerplate code from the persistence layer. Most of the code will then be in the parts of the application which require real complex persistence mechanisms.

1.5.2. Object Relational Mapping

Object-Relational Mapping (ORM) tools are achieving persistence of data with small side-effects on the domain objects while taking in charge transactions.

For this purpose, the supported framework is Hibernate. Moreover Spring Framework is advised to manage transactions as well as security, and connect them to the JavaEE container's relevant facilities (datasource and/or transaction manager).

The mapping is either available with XML configuration or JPA annotations.

As some non-standard features remain only available with XML or by using directly Hibernate, it is the supported default.

If these specific features are not needed by your application, you may use Hibernate with JPA annotations. The JPA container should not be considered as provided by the container.

For persistence that is not very object-related (batch updates or massive select requests for list displays) prefer the use of Spring's JdbcTemplates.

1.5.3. JDBC

JDBC is a Java SE API standardizing access and queries to relational databases. Using connection pools and datasources is required: direct connections to databases are not allowed. SQL requests have to be wrapped into either a PreparedStatement (from JDBC) or JdbcTemplates (from Spring Framework). The second choice is strongly advised because it manages exceptions and transactions automatically.

1.5.4. Datasources

Access to datasources should be done through JNDI.

Datasources' configuration should therefore be provided separately from the application, or documented.

1.5.5. Database

The European Parliament is mainly using Oracle (version in Annex) as a target database, and will possibly provide PostgreSQL as an alternative in the coming years.

Standard driver is the Oracle Thin driver (for example: `jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE`).

Persistence-related code must NOT be made specific to the underlying database.

1.6. INTEGRATION/REMOTING

1.6.1. JMS

Usage of Java Messaging System (JMS) within Java software development projects that need asynchronous communications between different components is the supported standard.

JMS provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages. This enables asynchronous communication for Java applications and can easily replace any software that relies on polling remote resources.

JMS defines a common set of enterprise messaging concepts and facilities. It attempts tominimize the set of concepts a Java language programmer must learn to use enterprise messaging products. It strives to maximize the portability of messaging applications.

The main concepts are:

- JMS Provider (think of it as the equivalent of a JDBC Driver for Messaging systems)
- JMS Messages (a series of well defined interfaces for Messages)
- JMS Domains (Point-to-point or Publish/Subscribe)

JMS must not be confused with Java Mail: JMS deals with Enterprise Messaging Systems through a unified API, Java Mail deals with emails.

Producing messages can be done whether through a Spring framework JMSTemplate or with classic JMS code.

Consuming messages should be done using Message Driven POJOs from Spring framework.

Use TextMessage object to exchange data such as XML or JSON.

Use of ObjectMessage object is forbidden and should be replaced by explicit serialization to a structured markup such as XML or JSON.

1.6.2. Email

Nowadays, most applications need to send emails. Usage of the JavaMail API to send emails from applications is the supported standard. Spring framework's Mail support is also supported as an abstraction for dealing with more intricate mails.

In order to avoid issues such as attached file blocked or messages detected as a SPAM, emails must comply with the following the rules:

- Emails must respect the standards to avoid being considered as malformed or as SPAM. Therefore, emails must conform to the different RFCs, in particular the RFC 822 which defines the mandatory and optional fields of an email.
- Sender's email addresses:
  - The sender's email address must be a real address, defined within the ActiveDirectory. It should be an application mailbox rather than a user mailbox. An application mailbox is longer available in time.
  - The sender's mailbox must be managed, because when a message cannot be delivered to its recipient, the NDR will be sent to this address. The most common way is to manage it through a delegation.
- Recipient's email address:
  - To send an email to a large amount of recipients, it must be sent in several batches.

Moreover, it is recommended to wait several minutes between the different sendings avoiding thus to be considered as a spammer and avoiding to overload the email infrastructure.

- To hide to the receivers, the list of all the people who receive the email, you can use the BCC (Blind Carbon Copy) field.

- Email content:
  - Emails must be encoded in UTF-8, thus emails can be written in all the European official languages.
  - For security reasons, it is forbidden to include scripting within emails.
- Attached files:
  - Size: attached files can be blocked by the SMTP gateway of the recipient's email address. Most of the time, the attached files are blocked if their size is bigger than 10 Mo.
  - Extension: attached files can be blocked for security reasons by the SMTP gateway of the receiver's email address. Some file formats can include viruses therefore some file extensions are blocked. So, when it is possible, it is recommended to send PDF or ZIP files.
  - Images & Background images: to send HTML emails including images or background images, it is preferable to use in-line images or to be sure that the source is accessible by the recipients (inside or outside)

1.6.3. Remoting and Services

In any remote access there are consumers and producers of some capabilities. The coupling between the two can theoretically be strong or loose.

To achieve robustness of the system and easy integration it is advised to choose looser coupling between components.

If both components are developed in the same time, consequence of strong coupling can be mitigated.

To provide a service conveys an obligation towards the consumers of the service, and therefore the design of a service MUST respect more characteristics:

- the service should provide a standardized service contract
- the service should be loosely coupled
- the service should be as stateless and reentrant as possible.

The use of remoting (without implying the exposition of a service with a standardized contract and Service Level) is an internal architectural decision that doesn't need to follow as strictly these design consideration.

Generally speaking, remoting protocols which exchange Serialized objects on the wire (such as Spring Remoting, RMI…) are forbidden. They MUST be replaced by REST web services.

If you want to provide a **service**, outside of any design consideration, you should consider **REST** web services.

REST is advised if the service is to be consumed directly by UI components (HTTP caching and content negotiation can be leveraged), and for most other web service scenarios.

1.6.4. REST Web Services

REST is the standard and supported way to implement API services in the European Parliament.

REST stand up for Representational State Transfer coined by Roy Thomas Fielding in *Architectural Styles and the Design of Network-based Software Architectures* [FieldingREST] in 2000.

It's difficult to have an application that follows completely the REST architectural style. Leonard Richardson developed a maturity model that explicitly measures the adherence to REST rules.

A blog post on the *REST Richardson Maturity Model by Martin Fowler* [FowlerMaturityModel] explains the different Levels very clearly.

Two frameworks may be used to develop REST services: Apache CXF REST framework and Spring REST stack based on Spring MVC.

1.6.5. Note on SOAP Web Services

SOAP Web Services are not standard anymore for the implementation of services in European Parliament. They are dealt as obsolescent technologies and described in the relevant obsolescence documentation.

1.7. CROSS-CUTTING CONCERNS

1.7.1. XML

1.7.1.1. XML handling

In order to manipulate XML directly from Java, the standard supported approach is to use thestandard JAXP APIs:

- DOM (Document Object Model)
- SAX (Simple API for XML)
- StAX (Streaming API for XML)
- TrAX (Transformation API for XML)

In doubt, for general purpose parsing/generation of XML, prefer StAX (supported implementation is Woodstox) for its simplicity of use and performance.

1.7.1.2. Object/XML Mapping (OXM)

Object/XML Mapping (OXM) is about converting an XML document to and from an object.

Object/XML Mapping enables applications to deal with the data defined in an XML document through an object model which represents that data.

The process of converting an object to an XML document is known as XML Marshalling, or XMLSerialization.

The process of converting an XML document to an object is known as XML Unmarshalling, or XMLDeserialization.

The conversion may require a precise mapping between the XML structure and the object's class definition. Defining such an explicit mapping is known as XML data binding.

Spring OXM adds an abstract layer in front of OXM frameworks, recommended instead of direct

use of XML APIs.

The main binding frameworks compatible with Spring OXM are Castor, JAXB2, JiBX and XStream.

1.7.2. Workflow

For the needs of application workflows, Activiti is the standard supported workflow engine. It can be easily embedded in an application through its integration with Spring framework. It makes it very easy to test (unit, integration) workflows and to use test doubles for that purpose. Activiti is based on the BPMN 2.0 standard, though this should only be considered implementation detail. Applications should not rely on BPMN-related standards and specifications in their design.

1.7.3. JSON

For any serialisation from or to JSON the Jackson library should be used.

1.7.4. Caching strategy

Usage of caches in order to improve performance of time-consuming data retrieval operations is standard.

EhCache is supported.

If you don't want to explicitly use cache in your business code or if you want to apply cross-cutting cache strategies to your application, AOP can be a good choice. Use of AOP is only recommended and supported if you do it through the Spring framework.

Usage of a cache is indicated especially for the following scenarios:

- Accessing a common read-only data-store
- Accessing remote services
- Optimizing repeating tasks

The cache facility should be implemented in such a way that it conforms to the following best practices:

- It should be possible to retrieve each element in the cache by a unique ID.
- Using a Tree cache allows to have different cache spaces for different parts of your application (therefore allowing to have different cache policies for a same element in different layers/functionalities of your application).
- Your application should not fail on cache errors.
- Your application should check for an element in the cache, and if it is not available, do the usual retrieval and put it in the cache before returning it.

In conclusion:

- During development/tests: use whichever simple implementation you see fit.
- During runtime and for production: prefer EhCache.

1.7.5. Applications security

Applications should handle Authentication using JAAS mechanisms and container authentication.

Authorization may be based on JAAS roles, or use a specific local mechanism depending on the

needs for global identity provisioning.

Deliverables developed externally should stick with BASIC authentication and JAAS. When deployed inside the European Parliament, BASIC can then easily be replaced by multiple other internal mechanisms without touching the application's code and affecting its general behaviour. Use of third-party libraries (like Spring Security) should be avoided for applications developed externally.

1.7.6. Testing strategy

Unit tests should be provided for key functions of the deliverables. A level of coverage should be agreed and respected.

JUnit 4 and above can be used. The use of any other testing framework should be fully documented.

For externally developed deliverables, Test doubles (such as Mocks, Stubs, Fakes and Dummies) are highly recommended in the case of interaction with resources available inside the European Parliament only. They may be provided by the internal teams at project inception for example.

Tests should be made reproducible, and should ideally be made to run through a build tool.

1.7.7. Logging

Logging should be done using slf4j as a default, and possibly commons-logging as a fallback if direct use of slf4j causes some issues with other used frameworks.

1.7.8. Configuration

Configuration in applications is conceptually separated in two parts:

- Application behaviour and frameworks configuration
- Containers and interfaces configuration

The second one refers to the configuration of any parameter that may vary depending on the environment surrounding an application.

This kind of parameter should therefore be **externalized**.

For example, any reference to a port, URL, tuning parameter (cache eviction delay for example), security credentials,…etc; should be externalized, in order to allow for a simpler adaptation of configuration when deployed inside the European Parliament's infrastructure.

European Parliament's application servers all have an etc/ folder put in the classpath where this kind of configuration files lies.

Inside of an application you may access such files using: `Thread.currentThread().getContextClassloader().getResource...()` … or even better, rely on the functionalities of frameworks dealing with this properly (property place holders in Spring Framework for example).

1.8. APPENDICES

1.8.1. Library/Technology versions

Here you'll find a list of library and technology versions as an indication of the current versions used in the European Parliament:

Library/Technology	Version
Java SE	1.8.0_x
Java EE	6.0
Spring Tools Suite	4.x
Eclipse	4.x
Apache Maven	3.x
Apache Tomcat	8.0.x
ActiveMQ Artemis (a.k.a. HornetQ 3+)	1.x
Spring	4.x
Ext-JS	6.x
JQuery	2.x
Angular	2.x
Hibernate	4.x
Oracle	11g
Apache CXF	3.x
Servlet	3.1
EhCache	2.x
JUnit	4
Activiti	5.x
slf4j	1.6

1.8.2. References

Here you'll find a list of external URLs pointing to documentation of technologies or tospecifications definitions.

Library/Technology	URL
Java SE	http://www.oracle.com/technetwork/java/javase/
Java EE	http://www.oracle.com/technetwork/java/javaee/
Spring Tools Suite	https://spring.io/tools
Apache Maven	http://maven.apache.org/
Apache Tomcat	http://tomcat.apache.org/
Subversion / SVN	http://subversion.apache.org/
Git	https://git-scm.com/
Spring	https://projects.spring.io/spring-framework/
Ext-JS	http://www.sencha.com/products/extjs/
Angular	https://angular.io/

Library/Technology	URL
JQuery	http://jquery.com/
POJO	http://en.wikipedia.org/wiki/Plain_Old_Java_Object
Hibernate	http://hibernate.org/
JDBC	http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html
Oracle Database	https://www.oracle.com/database/
CXF	http://cxf.apache.org/
Jax-RS	http://jcp.org/en/jsr/detail?id=311
Castor	https://castor-data-binding.github.io/castor/
JAXB2	http://www.oracle.com/technetwork/articles/javase/index-140168.html
JiBX	http://jibx.sourceforge.net/
XStream	http://x-stream.github.io/
EhCache	http://ehcache.org/
JAAS	http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html
JUnit	http://www.junit.org/
slf4j	http://www.slf4j.org/

- [FieldingREST] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- [FowlerMaturityModel] http://martinfowler.com/articles/richardsonMaturityModel.html