



## **Devonfw Guide**

v2.2.0

Copyright © 2015-2017 the Devonfw Team, Capgemini

# Table of Contents

Document Description .....	ix
1. Introduction to Devonfw .....	1
1.1. Devonfw, the package .....	1
2. Devcon User Guide .....	4
2.1. Requirements .....	4
2.2. Download Devcon .....	4
2.3. Devcon structure .....	5
2.3.1. Basic example .....	7
2.3.2. Parameters .....	7
2.3.3. Global parameters .....	7
2.4. Environment Setup .....	11
2.4.1. Prerequisites .....	11
2.4.2. Download .....	11
2.5. The Devon IDE .....	20
2.5.1. Introduction .....	20
2.5.2. Cobigen .....	20
2.5.3. IDE Plugins: .....	21
2.6. Sample Application .....	33
2.7. Devonfw Distribution Structure .....	42
2.7.1. conf .....	43
2.7.2. doc .....	43
2.7.3. scripts .....	43
2.7.4. settings .....	43
2.7.5. software .....	44
2.7.6. system .....	44
2.7.7. workspaces .....	44
2.7.8. console.bat .....	44
2.7.9. create-or-update-workspace.bat .....	44
2.7.10. EclipseConfigurator.log .....	44
2.7.11. eclipse-project.bat .....	44
2.7.12. update-all-workspaces.bat .....	45
2.7.13. variables.bat .....	45
2.8. Create New Application .....	46
2.8.1. From Eclipse .....	47
2.9. OASP4J Application Structure .....	51
2.9.1. Core Module .....	51
2.9.2. Server Module .....	55
2.10. Understanding the OASP4J and Spring Boot Configuration .....	56
2.10.1. Batch configuration .....	60
2.10.2. WebSocket configuration .....	60
2.10.3. Externalized Configuration .....	60
2.11. Database Configuration .....	63
2.11.1. Add dependencies .....	63
2.12. Set up and maintain database schemas with Flyway .....	68
2.13. Bean-Mapping using Dozer .....	73
2.13.1. Bean-Mapper Dependency .....	74
2.13.2. Bean-Mapper Usage .....	74

2.14. Write Unit Test Cases .....	76
2.14.1. Unit Test .....	76
2.15. Logging and Auditing .....	82
2.15.1. Logging .....	82
2.16. Data-Access Layer .....	86
2.16.1. Persistence .....	86
2.16.2. Database Configuration .....	100
2.16.3. Security .....	101
2.17. CRUD operations and DAO implementation .....	103
2.17.1. Validation .....	107
2.17.2. Define querys .....	108
2.17.3. DTO conversion .....	118
2.18. Getting Started Cobigen .....	124
2.19. Transfer-Objects .....	133
2.19.1. Business-Transfer-Objects .....	133
2.19.2. Service-Transfer-Objects .....	134
2.20. Logic Layer .....	135
2.20.1. Component Interface .....	135
2.20.2. Component Implementation .....	135
2.20.3. Passing Parameters Among Components .....	136
2.20.4. Security .....	137
2.21. Service Layer .....	138
2.21.1. Versioning .....	138
2.21.2. Interoperability .....	139
2.21.3. Protocol .....	139
2.21.4. Service Considerations .....	151
2.21.5. Security .....	151
2.22. Deployment on Tomcat (Client/Server) .....	153
3. Devonfw CookBook .....	158
3.1. The Reporting module - Report generation with JasperReports .....	158
3.1.1. Basic implementation .....	159
3.1.2. Subreports .....	162
3.2. The Winauth-AD module .....	165
3.2.1. Service CurrentUser .....	169
3.3. The Winauth-SSO module .....	170
3.4. The i18n module .....	175
3.4.1. Introduction .....	175
3.5. Devon-locale module(i18n resource converter) .....	181
3.5.1. Introduction .....	181
3.6. The async module .....	183
3.6.1. Introduction .....	183
3.7. The Integration Module .....	187
3.7.1. Introduction .....	187
3.8. Compose for Redis module .....	208
3.8.1. Introduction .....	208
3.8.2. Basic implementation .....	209
3.8.3. Available operations .....	209
3.9. Security .....	211
3.9.1. Authentication .....	211

3.9.2. Authorization .....	213
3.10. Web Services (JAX WS) .....	221
3.10.1. SOAP Testing .....	225
3.11. Batch Layer .....	226
3.11.1. Batch architecture .....	226
3.11.2. Implementation .....	228
3.12. Integrating Spring Data in OASP4J .....	247
3.12.1. Introduction .....	247
3.12.2. References .....	256
3.13. WebSocket .....	257
3.13.1. WebSocket configuration .....	257
3.14. File Upload and Download .....	259
3.14.1. Produces Annotation .....	259
3.14.2. Consumes Annotation .....	261
3.14.3. MIME Types .....	262
3.15. Microservices in Devonfw .....	263
3.15.1. Introduction .....	263
3.15.2. Microservices schema .....	264
3.16. Docker .....	278
3.16.1. Introduction .....	278
3.16.2. Fabric8io plugin .....	279
3.16.3. Docker tips .....	280
3.17. Production Line for Beginners (Practical Guide) .....	281
3.17.1. 2. Jenkins .....	281
3.17.2. 3. Nexus .....	284
3.18. Production Line .....	288
3.18.1. Introduction .....	288
3.19. Devon in Bluemix .....	293
3.19.1. Introduction .....	293
3.19.2. Bluemix Services .....	293
3.20. Deployment on Wildfly .....	297
3.21. Deploy oasp4j application to WebSphere Liberty .....	303
3.22. Cobigen advanced use cases: SOAP and nested data .....	308
3.22.1. Introduction .....	308
3.23. Compatibility guide for JAVA and TOMCAT .....	310
3.23.1. Using JAVA7 .....	310
3.23.2. Using java8 .....	311
3.23.3. Using Tomcat8 .....	311
3.24. Introduction to Devon4sencha .....	312
3.24.1. The universal application .....	313
3.25. Legal considerations .....	313
3.26. Sencha Cmd .....	314
3.27. Workspaces .....	314
3.27.1. Sencha packages .....	315
3.27.2. Sencha reloading .....	315
3.28. Application architecture .....	316
3.28.1. MVC (Model, View, Controller): .....	316
3.28.2. MVVM (Model, View, Controller, ViewModel, ViewController): .....	316
3.29. Structure of a Devon4sencha application .....	317

3.29.1. Model .....	317
3.29.2. Store .....	317
3.29.3. View .....	317
3.29.4. Controller .....	318
3.29.5. ViewController .....	319
3.29.6. ViewModel .....	320
3.30. Universal applications .....	320
3.31. Devon4sencha Sample application .....	322
3.31.1. Running the sample restaurant application .....	322
3.32. Creating a new application .....	326
3.33. Project Layout .....	331
3.34. Main Template .....	331
3.34.1. Displaying pages .....	333
3.35. Code conventions .....	335
3.35.1. Controllers, ViewModels and ViewControllers .....	335
3.35.2. Internationalization .....	335
3.35.3. Event Names and listeners .....	335
3.35.4. Code style .....	335
3.36. Sencha Client and Sencha Architect project generation .....	336
3.36.1. Getting ready .....	336
3.36.2. Generating .....	337
3.36.3. Deploying .....	338
3.37. Creating a new page .....	339
3.37.1. Step 1: ViewModel .....	339
3.37.2. Step 2: ViewController .....	339
3.37.3. Step 3: View .....	339
3.37.4. Step 4: MenuItem .....	340
3.37.5. Step 5: Controller .....	340
3.38. Create a CRUD page .....	342
3.38.1. CRUD Packaging .....	342
3.38.2. Step 1: Add a CRUD page to the application menu .....	342
3.38.3. Step 2: Controller, a view factory and REST endpoints definition .....	343
3.38.4. Step 3: Create a model .....	344
3.38.5. Step 4: Create the Store .....	344
3.38.6. Step 5: Create the view and viewController .....	345
3.38.7. Step 6: Create i18n literals .....	346
3.39. Complete CRUD example (Create, Read, Update and Delete) .....	347
3.40. Extending the CRUD .....	354
3.41. Extending the CRUD .....	354
3.41.1. Refreshing the grid after changing some tables .....	354
3.41.2. Double-click functionality .....	355
3.42. Extending the CRUD: ViewModels .....	357
3.43. Pagination .....	365
3.44. Extending the CRUD: searching and filtering tables .....	367
3.45. Extending the CRUD: Adding an inbox to search tables .....	370
3.46. Extending the CRUD: Change the state of a table .....	376
3.47. Extending the CRUD: Editing menu order .....	379
3.47.1. Create Edit Order View .....	379
3.47.2. Drag and drop .....	383

3.48. Extending the CRUD: Working with layouts .....	386
3.49. Extending the CRUD: Grid editable .....	390
3.49.1. Creating the grid .....	390
3.49.2. Cellediting plugin .....	392
3.49.3. Rowediting plugin .....	395
3.49.4. Add empty records .....	395
3.50. Extending the CRUD: Custom plugin .....	397
3.51. The class system .....	399
3.51.1. Ext .....	399
3.52. Components .....	399
3.52.1. Creating Custom Components .....	399
3.53. Rest endpoints .....	405
3.53.1. URL template resolution .....	406
3.53.2. Query string .....	406
3.53.3. Alternative PUT format .....	406
3.53.4. Automatic JSON response decode .....	407
3.53.5. Base Url configuration .....	407
3.54. Sencha data package .....	408
3.54.1. Devon.restproxy .....	408
3.54.2. Pagination .....	409
3.54.3. Sorting .....	409
3.55. Ext.window.Window .....	411
3.55.1. Basic Window .....	411
3.55.2. Conclusion .....	417
3.56. Layouts and Containers .....	418
3.56.1. Containers .....	418
3.57. Layouts .....	418
3.57.1. Using Layouts .....	419
3.58. Grids .....	420
3.58.1. Basic Grid Panel .....	420
3.59. Forms .....	422
3.59.1. Basic Form Panel .....	422
3.60. CORS support .....	423
3.60.1. Configure CORS support .....	423
3.60.2. Bypass CORS security check during development .....	423
3.61. Security .....	426
3.61.1. Login process .....	426
3.61.2. User security info .....	427
3.61.3. Controlling visibility of controls based on user roles .....	427
3.62. Theming .....	428
3.62.1. Configuring Theme Inheritance .....	428
3.62.2. Using a Theme in an Application .....	429
3.62.3. Making changes in the Theme .....	430
3.62.4. Adding customs UI's .....	432
3.63. Error processing .....	434
3.64. Internationalization .....	436
3.64.1. Configuration .....	436
3.64.2. Message bundles .....	436
3.64.3. Change language .....	437

4. Devon4Sencha Cookbook .....	438
4.1. Mocks with Simlets: simulating server responses .....	438
4.1.1. Simlet Service .....	438
4.1.2. JSON Simlet .....	438
4.2. Simlets in Devon4Sencha .....	439
4.2.1. Use Simlets in your client application .....	441
4.3. Simlets Benefits .....	441
4.4. Devon4Sencha best practices .....	442
4.4.1. Bad formed code .....	442
4.4.2. Excessive or unnecessary nesting of component structures .....	442
4.4.3. Memory leaks caused by failure to cleanup unused components .....	443
4.4.4. Poor folder structure for source code .....	445
4.4.5. Use of global variables .....	446
4.4.6. Use of <i>id</i> .....	447
4.4.7. Unreliable referencing of components .....	448
4.4.8. Failing to follow upper or lowercase naming conventions .....	448
4.4.9. Making your code more complicated than necessary. ....	449
4.4.10. Nesting callbacks are a nightmare .....	449
4.4.11. Caching and references .....	450
4.4.12. Indentation .....	451
4.4.13. One class per file .....	452
4.4.14. Too much work to return .....	452
4.4.15. Comments or Documentation .....	453
4.4.16. Operators .....	454
4.4.17. Be lazy .....	454
4.4.18. Knowing this .....	455
4.4.19. Additional resources .....	455
4.5. Devon4Sencha Tools .....	456
4.5.1. Code Analysis Tools .....	456
4.5.2. Analysing code with Sencha Cmd .....	456
4.5.3. Sonar for JavaScript .....	456
4.6. How to do effective Devon4Sencha code reviews .....	458
4.6.1. Benefits .....	458
4.6.2. Best practices .....	458
4.6.3. How to handle code reviews .....	459
4.6.4. On mindset .....	459
4.6.5. Code review tools .....	459
4.7. Devon4Sencha testing tools .....	461
4.7.1. Tools for testing your Sencha code .....	461
4.7.2. Sencha Test .....	461
4.8. Adapting devon4sencha apps to microservices .....	463
4.8.1. Introduction .....	463
4.8.2. Security changes .....	463
4.9. How to start a cordova project from a sencha project .....	465
4.9.1. Recommended readings: .....	465
4.9.2. Minimum requirements: .....	465
4.9.3. Steps to do on the Sencha project .....	465
4.9.4. Steps to do on the Cordova project .....	467
4.9.5. Steps to run the Sencha project in the Cordova project .....	468

---

4.9.6. OASP4Js .....	469
4.9.7. Oasp4js tutorial .....	472
5. Appendix .....	473
5.1. Frequently Asked Questions (FAQ) .....	473
5.1.1. Server .....	473
5.1.2. Client .....	474
5.1.3. Configuration issues .....	475
5.1.4. Crosscutting concerns .....	478
5.2. Working with Git and Github .....	479
5.2.1. Issue creation .....	487
5.2.2. Code Contribution .....	488
5.3. Devcon Command Reference .....	489
5.3.1. dist .....	489
5.3.2. doc .....	492
5.3.3. github .....	492
5.3.4. help .....	494
5.3.5. oasp4j .....	494
5.3.6. oasp4js .....	497
5.3.7. project .....	500
5.3.8. sencha .....	505
5.3.9. workspace .....	507
5.3.10. system .....	508
5.4. Devcon Command Developer's guide .....	509
5.4.1. Introduction .....	509
5.4.2. Conclusion .....	524
5.5. Devon module Developer's Guide .....	525
5.5.1. Usage .....	527
5.6. Steps to use Devon distribution in Linux OS .....	532
5.7. List of Components .....	533
5.8. Devonfw Release notes 2.2 "Courage" .....	541
5.8.1. OASP4js 2.0 .....	541
5.8.2. OASP4j 2.4.0 .....	543
5.8.3. Microservices Netflix .....	543

# Document Description

[Devonfw](#) provides a solution to building applications which combine best-in-class frameworks and libraries as well as industry proven practices and code conventions. It massively speeds up development, reduces risks and helps you to deliver better results.

This document contains the complete compendium of the [Devonfw](#). From this link you will also find the latest release or nightly snapshot of this documentation.

**Current versions:**

- DevonfwGuide - v2.2.0
- Devonfw - v2.2.0

**Authors:**

[Devonfw](#) Team, Capgemini

# 1. Introduction to Devonfw

[[getting-started-introduction-to-devonfw\_welcome-to-devonfw]] == Welcome to Devonfw



Welcome to **Devonfw**, the Devon Framework. This is a product of the CSD industrialization effort to bring a standardized platform for custom software development within Capgemini APPS2. This platform is aimed at engagements where clients do not force the use of a determined technology so we can offer a better alternative coming from our experience as a group.

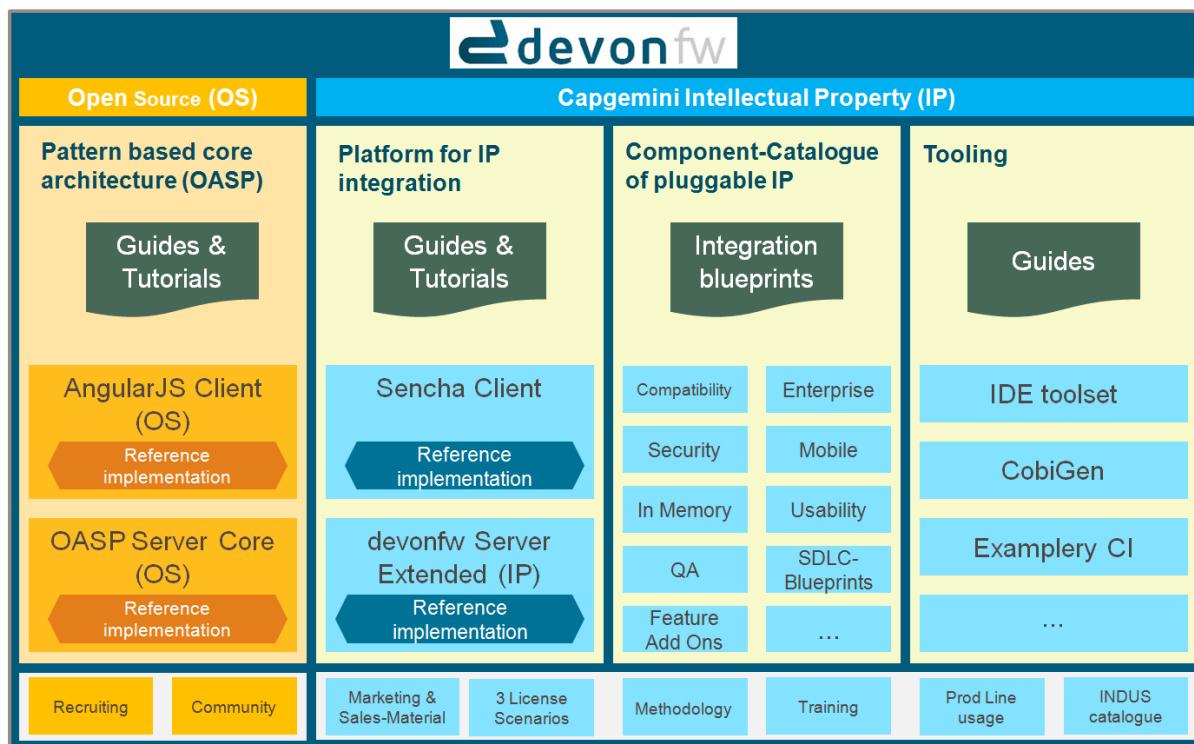
The current version of the framework is oriented at development of web applications based on the Java EE programming model using the Spring framework project as the default implementation. The main idea is not to create a monolithic framework but to offer helpers to develop applications with proven patterns. Devonfw tries to be close to standards and not relying too much on custom implementations. In this way, the ideas can be reused on other implementations such as on a Java EE pure solution (that is, with no Spring involved). Furthermore Devonfw comes with a series of development guides, tutorials and a reference application you can derive from. Devonfw is making extensive use of standards and existing products and not relying too much on technical custom components implemented by CG.

## 1.1 Devonfw, the package

Devonfw contains:

- Implementation blueprints for a modern cloud-ready server and a choice on JS-Client technologies (either open source AngularJs or a very rich and impressive solution based on commercial Sencha UI).
- Quality documentation and step-by-step tutorials.
- Highly integrated and packaged development environment based around Eclipse and Jenkins. You will be ready to start implementing your first customer-specific use case in 2h time.
- Iterative eclipse-based code-generator that understands "Java" and works on higher architectural concepts than Java-classes.
- Example application as a reference implementation.
- Support through large community + industrialization services (Standard Platform as a service) available in the iProd service catalog.

The following picture shows main building blocks of the platform.



Devonfw uses a state-of-the-art open source core reference architecture for the server (today considered as commodity in the IT-industry) and on top of it an ever increasing number of high-value assets that are Capgemini IP.

[[getting-started-introduction-to-devonfw\_why-do we use an open source core]] == Why do we use an open source core

In some countries of the SBU clients only accept open source stacks (most to mention customers who want to stay vendor-independent by principle), so being open is a market need. This also is the rationale for providing a open source reference architecture for the client built with AngularJs (open source itself). The picture above shows the open-source parts of devonfw on the left.

[[getting-started-introduction-to-devonfw\_devonfw-and open source]] == Devonfw and Open Source

Devonfw is composed of an Open Source part that can be freely used by other people and proprietary addons which are Capgemini IP and can be used only on Capgemini engagements.

The Open Source part of Devonfw is called OASP (Open Application Standard Platform) which started as an initiative of the German BU (Apps evolve). This project offers both a server side solution (based on Spring) and an AngularJS solution for client side.

Having an Open Source foundation is an advantage on clients that don't want to be tied to an internal framework of a company like Capgemini. Since all the code is provided, the client is free to evolve it freely.

OASP itself is based on other established Open Source framework such as Spring, CXF or Hibernate. Adopting best practices and being enriched by the experience of our community of experts.

[[getting-started-introduction-to-devonfw\_contributing-to devonfw]] == Contributing to Devonfw

Devonfw Platform is organized in a way that it is easy for you to contribute. Therefore we have chosen to use *github* (the number one platform for social coding) which provides lean processes and great tooling.

---

Currently most repositories are marked as private, hence invisible to you. You can send an email from your Capgemini account with your *github* login to the devonfw team if you want to get access. Please ensure your real name is set in your *github* account or your login is matching your Capgemini CORP login. You will be also added to our Capgemini OASP mailing list (see contact).

In order to contribute code we use *git* and *github pull-requests*. Lead developers can directly commit to the git repository while (later) everybody can clone and fork the repository and create *pull-requests*. These can be reviewed, commented and discussed and finally integrated (or rejected).

We are very happy to receive contributions from projects or individual experts. Before you invest your time and work into a larger change or contribution please get in contact before, to ensure that you will not waste your energy (somebody else might already work on the same thing, etc.). To get in touch and discuss with us please meet us in *Yammer*.

[[getting-started-introduction-to-devonfw\_devonfw-backend tutorial]] == Devonfw backend tutorial

[Oasp4j tutorial](#): You can follow a complete tutorial for the Devonfw backend implementation based on *Oasp4j*. In this tutorial you will see an overview of the backend technologies and how to implement a basic (but fully functional) backend application based on Devonfw.

## 2. Devcon User Guide



The Devon Console, **Devcon**, is a cross-platform command line tool running on the JVM that provides many automated tasks around the full life-cycle of Devon applications, from installing the basic working environment and generating a new project, to running a test server and deploying an application to production.

Devcon is the easiest way to use Devonfw. By accompanying on project automation, easy command execution and declarative configuration, it gets out of your way and let you focus on your code.

### 2.1 Requirements

You will need to have a Java JDK 1.7 or 1.8 installed on your system (devcon GUI requires JDK 1.8). It is **not** necessary to use the Devonfw Distribution as Devcon can be used independently from that environment. You might even want to use Devcon itself to download & extract the Devon Distribution zip file to and on your system! (see down...)

### 2.2 Download Devcon

From the Devonfw version 2.1.1 onwards Devcon is included by default within the Devonfw distributions.

However, if you want to install Devcon locally (Devcon can be used to get the last version of Devonfw distribution, so it can be useful to do it that way) you will need to complete the following steps:

- [\*\*download the Devcon jar from the devonfw repository\*\*](#)
- After downloading, open console or command prompt (on Windows: CMD or Powershell), navigate ("cd") to the path where you've downloaded the jar file and execute the following command:

```
java -jar devcon.jar system install
```

This will install the devcon binaries to a folder '.devcon' in your '%HOME%' directory. Devcon is now added to the '%PATH%' environment variable on your system, allowing it to execute directly from the console. However, after devcon installation on Windows, you need to close the console and reopen it in order to work the environment variables (only for the first time).

Remember that those steps are only necessary in case you want to install Devcon locally on your environment. As mentioned before an installation of Devcon is included by default in the Devonfw distributions (version 2.1.1 or higher) which means that executing the `console.bat` script will open a command line with Devcon available.

Once you know how obtain Devcon, two commands are available to open devon user guide. Try either one, or both of

```
devon doc userguide  
devcon doc userguide
```

to open up the Devcon user guide in your default system web browser. The commands 'devon' and 'devcon' can be used interchangeably.

## 2.3 Devcon structure

Devcon is based on three basic elements:

- modules
- commands
- parameters

So Devcon is a tool based on modules where each module groups several commands related to the module functionality and each command may need parameters to work in one way or another. Each command is used to accomplish one task and may need some parameters in order to achieve its goal. These parameters can be mandatory or optional. The mandatory parameters must be provided by the user when launching the command and the optional parameters can be provided by the user in the same way but if not, they can be read from a configuration file. Apart from this, we can use global parameters that are independent from the modules and commands and that will be helpful in order to obtain Devcon information (like help) or other basic configuration features, these parameters will be explained later.

[[devcon-user-guide\_devcon-basic usage]] == Devcon basic usage

There are two ways to run Devcon:

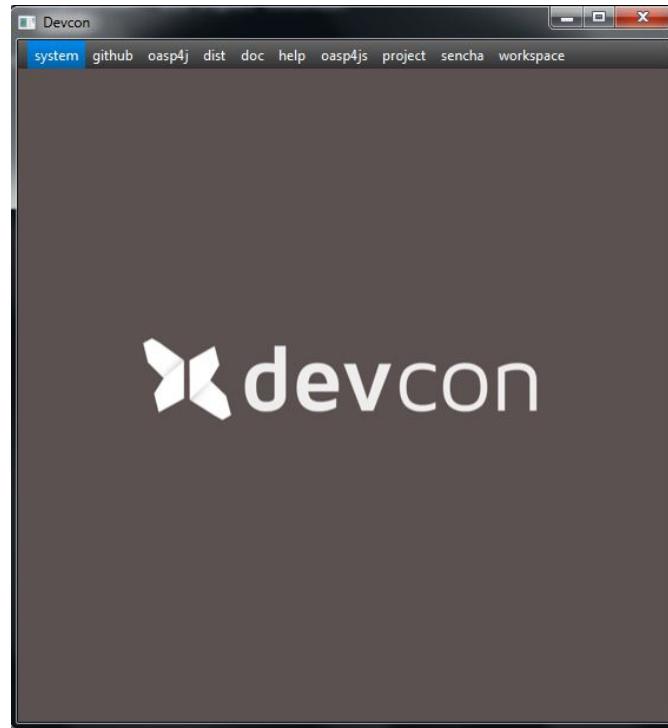
- Using Devcon GUI
- Using command prompt

[[devcon-user-guide\_using-devcon gui]] === Using Devcon GUI

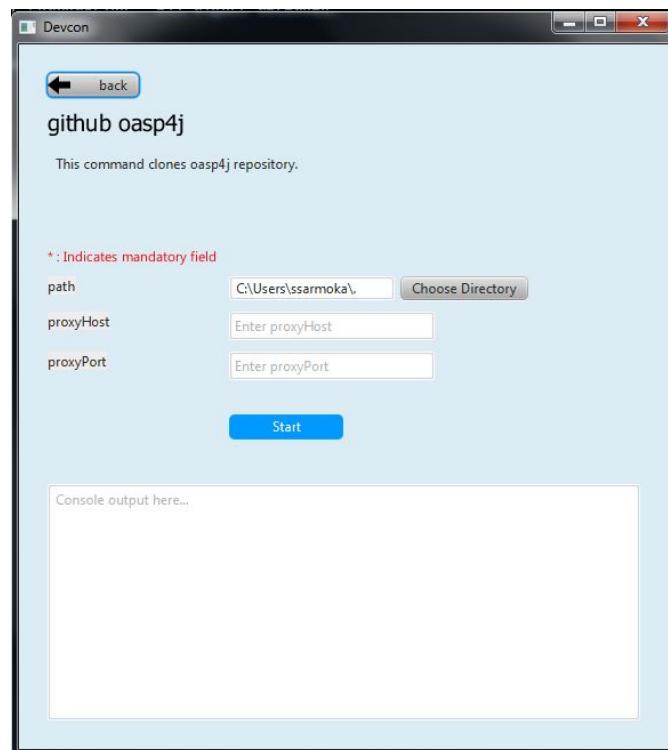
To run Devcon using the Devcon GUI, open command prompt and run below command (remember that **you will need JDK 1.8 to use Devcon's GUI**):

```
devon -g
```

The following application window will appear.



If you navigate to a particular module, it will show all the available commands from the respective module. You can find the "Exit" option to close the application, in the first menu on the left. Once you click on any command, the corresponding command form will be displayed on the screen. Refer following screenshot:



[[devcon-user-guide\_using-command prompt]] === Using command prompt

Accordingly with the defined structure, the devcon usage is based on the definition of each of its elements named : module, command and parameters. Therefore, in the command prompt, one must specify each of these elements in the correct order:

```
devon [module] [command] [parameters ...]
```

- the module will be the first word after the "devon" keyword.
- the command will be the second word after the "devon" keyword.
- the parameters are the rest of the elements defined after the command.

[[devcon-user-guide\_defining-the module and the command]] ===== Defining the module and the command

Both, module and command are defined by an identifier i.e. 'name' that the module or command have assigned in devcon.

[[devcon-user-guide\_defining-the parameters]] ===== Defining the parameters

The parameter definition is divided in two parts. The first one is the parameters identifier i.e. the *name* that the parameter has assigned within the devcon app **preceded by a single dash**. The second part of the parameter definition is the parameter value.

### 2.3.1 Basic example

Following is a basic example of a devcon using command prompt:

```
C:\>devon foo saySomething -message hello
```

where:

- **foo** is the module.
- **saySomething** is the command of the *foo* module to be executed.
- '**-message**' is the parameter that the command *saySomething* needs to be executed.
- **hello** is the value for the *message* parameter.

### 2.3.2 Parameters

As its mentioned before from the point of view of the commands, we have two types of parameters: the mandatory parameters and the optional parameters. The mandatory parameters must be provided by the user specifying the parameter identifier and the value in the command line. The optional parameters must be also provided to the app but, if the user do not specify it, devcon will use a default value for them.

### 2.3.3 Global parameters

Devcon handles a third type of parameter that has nothing to do with command parameters. We are referring it as *global parameters*.

The *global parameters* are a set of parameters that works in global context, which means it will affect the behaviour of the command in the first phase i.e. before launching the command module itself.

As these parameters act in a global context, we do not need to provide the values for them. They work as *flags* to define some internal behaviour of devcon.

In the current version of Devcon we have the following global parameters :

- global parameter *gui*: defined with **-g** or **--gui**
- global parameter *help*: defined with **-h** or **--help**.
- global parameter *prompt*: defined with **-p** or **--prompt**.
- global parameter *stacktrace*: defined with **-s** or **--stacktrace**.
- global parameter *version*: defined with **-v** or **--version**.

### **gui** parameter

As we saw earlier the global parameter *gui* (**-g**) is the way we will launch the Devcon's graphical user interface. So to complete that operation we only need to execute

```
devon -g
```

### **help** parameter

The global parameter *help* is very useful to show overall help info of Devcon or also for showing more detailed info of each module and command supported. For example, if you don't know anything about how to start with Devcon, the option **-h** (or **--help**) will show a summary of the devcon usage, listing the global parameters and the available modules alongside a brief description of each one.

```
C:\>devon -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: devon <>module><>command><> [parameters...]
Devcon is a command line tool that provides many automated tasks around
the full life-cycle of Devon applications.
-h,--help      show help info for each module/command
-v,--version   show devcon version
List of available modules:
> help: This module shows help info about devcon
> sencha: Sencha related commands
> dist: Module with general tasks related to the distribution itself
> doc: Module with tasks related with obtaining specific documentation
> github: Module to create a new workspace with all default configuration
> workspace: Module to create a new workspace with all default configuration
```

As a global parameter, if you use the **-h** parameter with a module, it will show the help info related to given module including a basic usage and a list of the available commands in given module.

```
C:\>devon foo -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: foo <>command><> [parameters...]
This is only a test module.

Available commands for module: foo
> saySomething: This command is for say something
```

In the same way, as a global parameter, if we use the **-h** parameter with a command, instead of launching the command the help info related to the command will be shown

```
D:\>devon foo saySomething -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: saySomething [-message] [-signature]
This command is to say something
```

```
-message      the message to be written
-signature    the signature
```

Even if you specify the needed parameters, the behaviour will be the same as we stated that the global parameters affect how devcon behaves before launching the commands

```
D:\>devon foo saySomething -message hello -signature John -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: saySomething [-message] [-signature]
This command is to say something
-message      the message to be written
-signature    the signature
```

### ***prompt* parameter**

With this global parameter, you can ask devcon to prompt for all parameters (both optional and mandatory) when launching a command.

To give an example, you can use the *oasp4j create* command (that creates a new server project based on *OASP4J* model). In this case we would need to provide several parameters so the command call would look like

```
D:\devon-dist>devon oasp4j create -servername myServer -groupid com.capgemini -packagename
com.capgemini.myServer -version 1.0
```

As you can see the command is defined by `devon oasp4j create` words and the rest of the command line attributes are parameters.

With the global parameter **-p** Devcon gives the user the option to avoid defining any parameter when launching the command and provide step by step all parameters after that, so the usage of some commands can be way easier.

Going back to the previous example if we use the **-p** parameter we get

```
D:\devon-dist>devon oasp4j create -p
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
Command: devon oasp4j create
Description: This command is used to create new server project

Parameter: serverpath - where to create
->
Parameter: servername - Name of project
-> myServer
Parameter: packagename - package name in server project
-> com.capgemini.myServer_
[...]
```

As you can see with the **-p** parameter Devcon asks for each parameter related to a command (the optional ones can be left blank as the *serverpath* in the example) and the user can provide them one on one, getting rid of the concern of knowing what parameters needs a command.

### ***version* parameter**

This is a simple option that returns the devcon running version and is defined with **-v** (or **--version**). As the *help* option this will show the devcon version even though we have defined a command with all required parameters.

```
D:\>devon -v
```

```
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
devcon v.1.0.0
```

```
D:\>devon foo saySomething -message hello -signature John -v
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
devcon v.1.0.0
```

[[devcon-user-guide\_first-steps with devcon]] == First steps with Devcon This section describes how to start using devcon from scratch. For this, you can use the global option `-h` (help) in order to figure out which commands and parameters you need to define. But in a very first approach, only the command `devon` will be enough. Therefore, the first step is to look for a module that fits your requirements. As mentioned above, you can do this with the `help` option (defined as `-h` or `--help`) or with a simple command `devon`. If you do not specify any information, you will see a summary of the general help information with an example of usage, a list with global parameters and the available modules.

```
D:\>devon
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: devon <<module>> <<command>> [parameters...]
Devcon is a command line tool that provides many automated tasks around
the full life-cycle of Devon applications.
-h,--help      show help info for each module/command
-v,--version   show devcon version
List of available modules:
> help: This module shows help info about devcon
> sencha: Sencha related commands
> dist: Module with general tasks related to the distribution itself
> doc: Module with tasks related with obtaining specific documentation
> github: Module to create a new workspace with all default configuration
> workspace: Module to create a new workspace with all default configuration
```

Once you have the list of modules and an example of how to use it, you may need to get the devon distribution to go deeper in module `*dist*`, for that you can again use the `help` option after the module definition.

```
D:\>devon dist -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: dist <<command>> [parameters...]
Module with general tasks related to the distribution itself

Available commands for module: dist
> install: This command downloads the distribution
> s2: Initializes a Devon distribution for use with Shared Services.
```

Now, you know that the `dist` module has two commands, the `install` command and the `s2` command and you can see a brief description of each one therefore you can decide which one you need to use. In case you have to get a devon distribution, it can be found by the `install` command with the `help` option.

```
D:\>devon dist install -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: install [-password] [-path] [-type] [-user]
This command downloads the distribution
  -password  the password related to the user with permissions to download
              the Devon distribution
  -path      a location for the Devon distribution download
  -type      the type of the distribution, the options are:
              'oaspide' to download OASP IDE
              'devondist' to download Devon IP IDE
  -user      a user with permissions to download the Devon distribution
```

So now you know that the *install* command of the *dist* module needs:

- user with permissions to download the distribution.
- the related password.
- the path where the distribution file must to be downloaded.
- the type of distribution that can be 'aspide' or 'devondist'.

With all the information, you can launch a fully functional command such as:

```
D:\>devon dist install -user john -password 1234 -path D:\Temp\MyDistribution -type devondist
```

Regarding the order of the command parameters, devcon will order them internally so that you don't have to concern about that point and you can specify them in the order you want. The only requirement is that all mandatory parameters should be provided.

[[devcon-user-guide\_-devcon command reference]] == Devcon command reference

For a full reference of all the available commands in Devcon, see the [Devcon Command Reference](#)

[[devonfwguide\_getting-started with the service layer]] == Getting started with the Service Layer

## 2.4 Environment Setup

In this section, you will learn how to setup the Devonfw environment and start working on first project based on Devonfw.

The Devonfw environment contains all software and tools necessary to develop applications with Devonfw.

### 2.4.1 Prerequisites

In order to setup the environment, following are the prerequisites:

- Internet connection (including details of your proxy configuration, if necessary)
- 2GB of free disk space
- The ZIP containing the latest Devonfw distribution

### 2.4.2 Download

Download latest Devonfw distribution from [our Teamforge site](#).

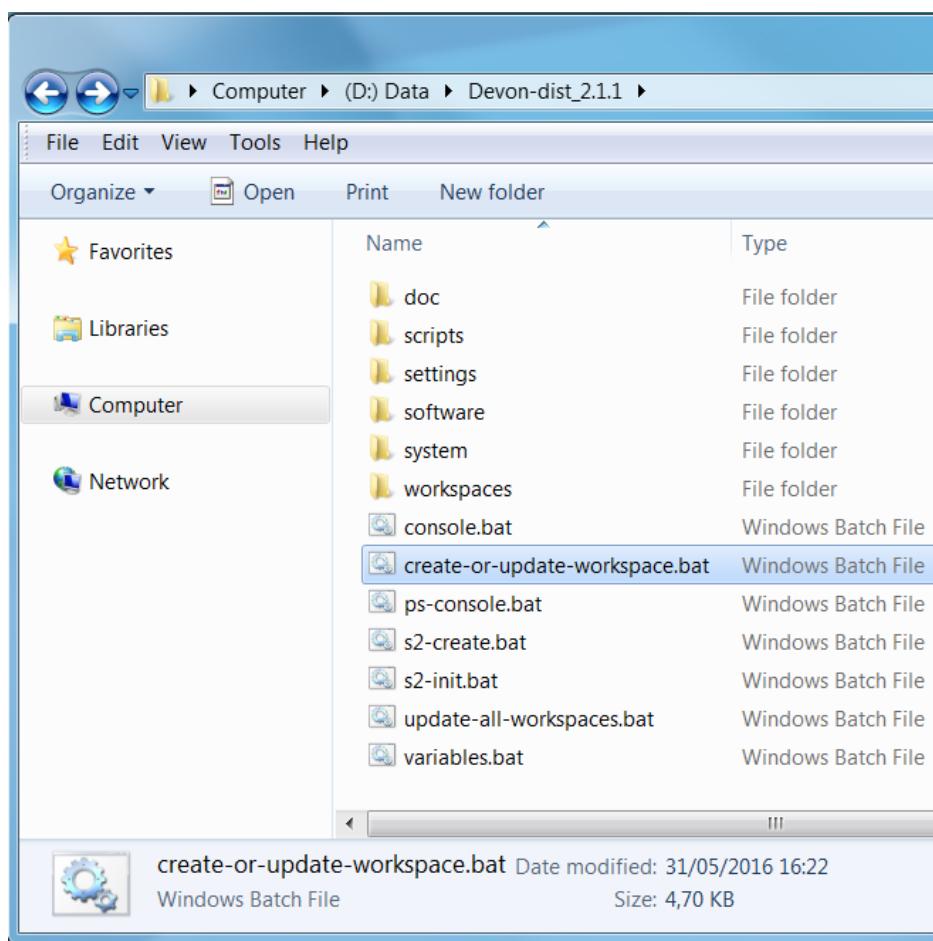


#### Using devcon

You can do it using devcon with the command `devon dist install` [learn more here](#) After a successful install you can initialize it with the command `devon dist init` [learn more here](#)

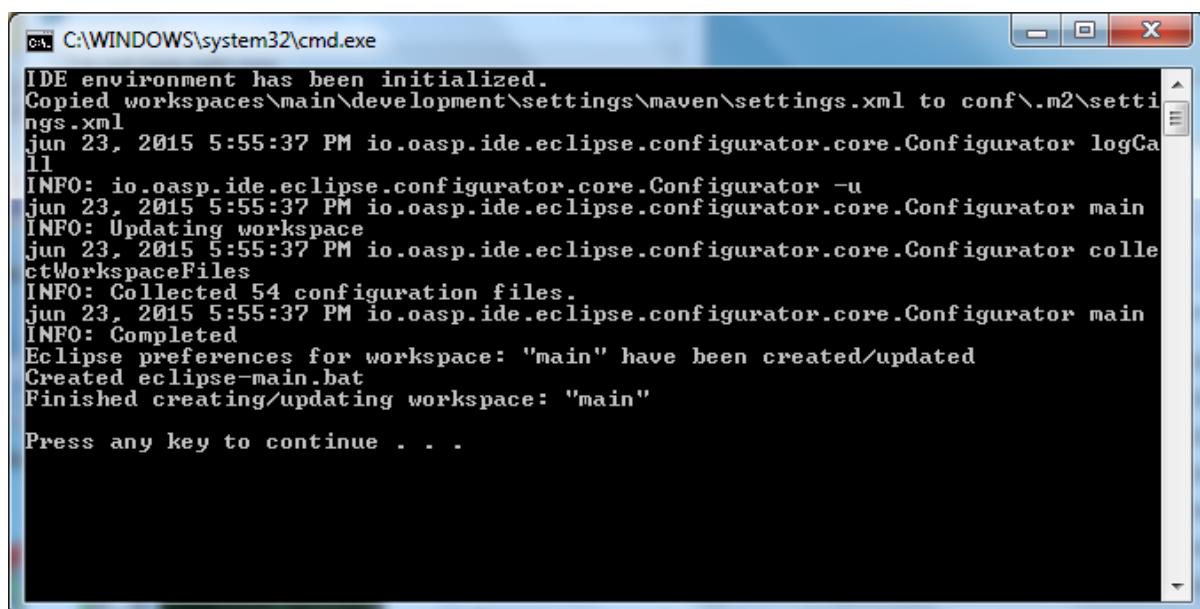
[[getting-started-download-and-install\_prepare-the workspace]] == Prepare the workspace

1. Unzip the Devonfw distribution into a directory of your choice. **The path to the Devonfw distribution directory should contain no spaces**, to prevent problems with some of the tools.
2. Run the batch file "create-or-update-workspace.bat".



This will configure the included tools like Eclipse with the default settings of the Devonfw distribution.

The result should be as seen below



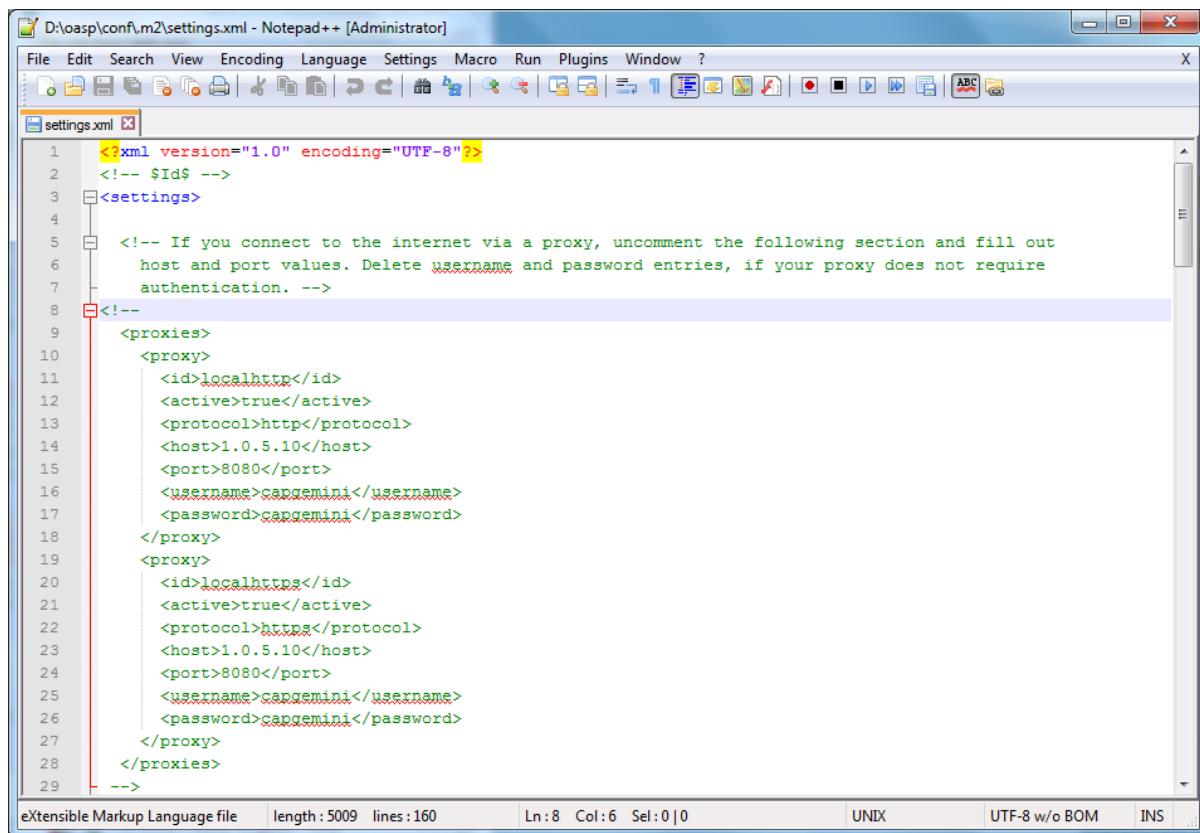
```
IDE environment has been initialized.
Copied workspaces\main\development\settings\maven\settings.xml to conf\.m2\settings.xml
Jun 23, 2015 5:55:37 PM io.oasp.ide.eclipse.configurator.core.Configurator logConfig
INFO: io.oasp.ide.eclipse.configurator.core.Configurator -u
Jun 23, 2015 5:55:37 PM io.oasp.ide.eclipse.configurator.core.Configurator main
INFO: Updating workspace
Jun 23, 2015 5:55:37 PM io.oasp.ide.eclipse.configurator.core.Configurator collectWorkspaceFiles
INFO: Collected 54 configuration files.
Jun 23, 2015 5:55:37 PM io.oasp.ide.eclipse.configurator.core.Configurator main
INFO: Completed
Eclipse preferences for workspace: "main" have been created/updated
Created eclipse-main.bat
Finished creating/updating workspace: "main"

Press any key to continue . . .
```

3. If you use a proxy to connect to the Internet, you have to manually configure it in the following tools:

### Maven

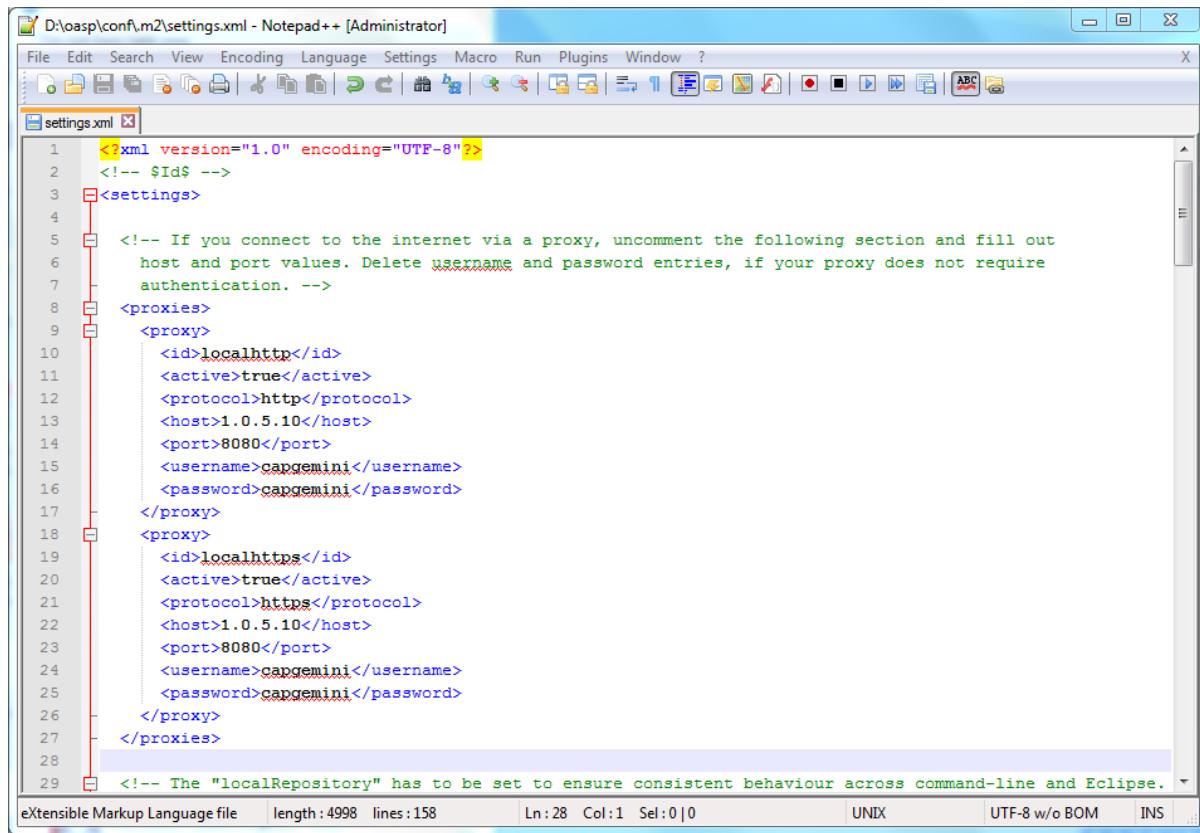
Open the file "conf/.m2/settings.xml" in an editor



```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id$ -->
<settings>
  <!-- If you connect to the internet via a proxy, uncomment the following section and fill out
       host and port values. Delete username and password entries, if your proxy does not require
       authentication. -->
  <!--
  <proxies>
    <proxy>
      <id>localhttp</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>1.0.5.10</host>
      <port>8080</port>
      <username>capgemini</username>
      <password>capgemini</password>
    </proxy>
    <proxy>
      <id>localhttps</id>
      <active>true</active>
      <protocol>https</protocol>
      <host>1.0.5.10</host>
      <port>8080</port>
      <username>capgemini</username>
      <password>capgemini</password>
    </proxy>
  </proxies>
-->
```

Remove the comment tags around the <proxy> section at the beginning of the file.

Then update the settings to match your proxy configuration.



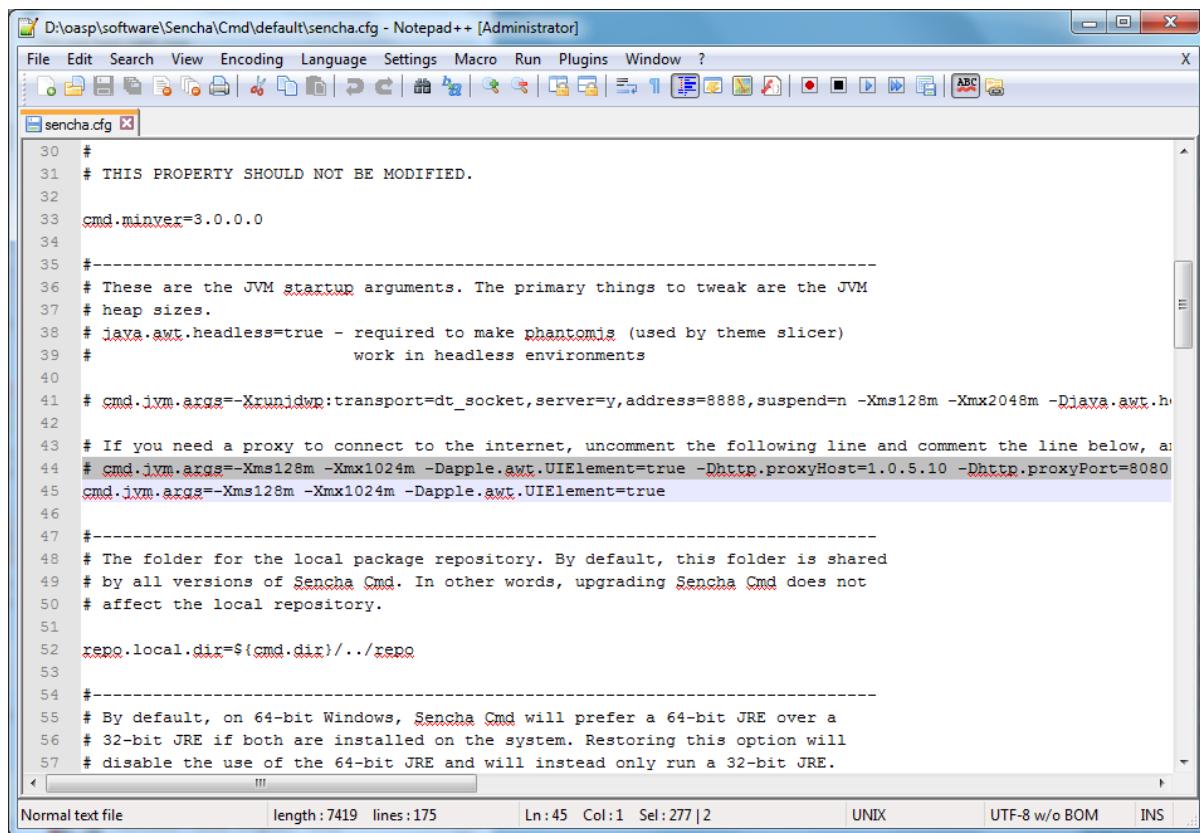
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- $Id$ -->
3 <settings>
4
5 <!-- If you connect to the internet via a proxy, uncomment the following section and fill out
6 host and port values. Delete username and password entries, if your proxy does not require
7 authentication. -->
8 <proxies>
9   <proxy>
10     <id>localhttp</id>
11     <active>true</active>
12     <protocol>http</protocol>
13     <host>1.0.5.10</host>
14     <port>8080</port>
15     <username>capgemini</username>
16     <password>capgemini</password>
17   </proxy>
18   <proxy>
19     <id>localhttps</id>
20     <active>true</active>
21     <protocol>https</protocol>
22     <host>1.0.5.10</host>
23     <port>8080</port>
24     <username>capgemini</username>
25     <password>capgemini</password>
26   </proxy>
27 </proxies>
28
29 <!-- The "localRepository" has to be set to ensure consistent behaviour across command-line and Eclipse. -->
```

eXtensible Markup Language file | length:4998 lines:158 | Ln:28 Col:1 Sel:0|0 | UNIX | UTF-8 w/o BOM | INS | ...

If your proxy does not require authentication, simply remove the <username> and <password> lines.

## Sencha Cmd

Open the file software/Sencha/Cmd/default/sencha.cfg in an editor

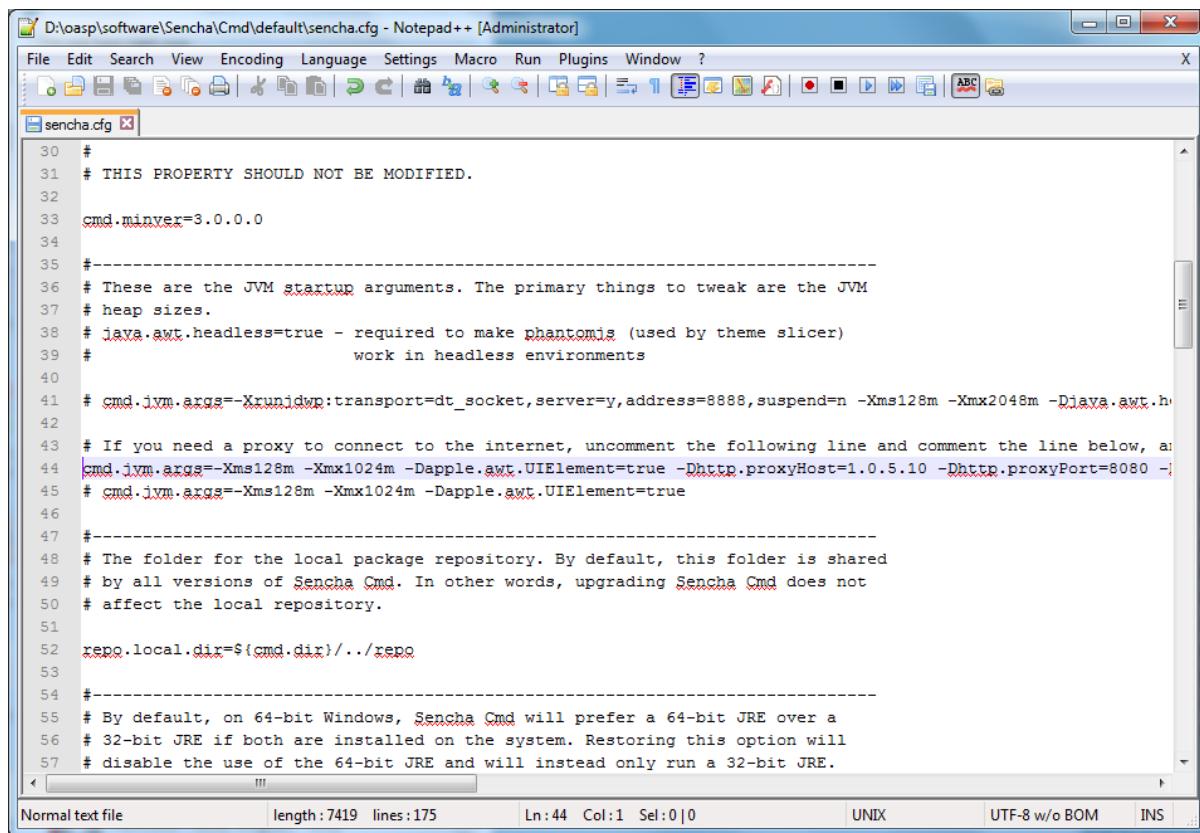


```
30 #
31 # THIS PROPERTY SHOULD NOT BE MODIFIED.
32
33 cmd.minver=3.0.0.0
34
35 #-----
36 # These are the JVM startup arguments. The primary things to tweak are the JVM
37 # heap sizes.
38 # java.awt.headless=true - required to make phantomjs (used by theme slicer)
39 #                         work in headless environments
40
41 # cmd.jvm.args=-Xrunjdwp:transport=dt_socket,server=y,address=8888,suspend=n -Xms128m -Xmx2048m -Djava.awt.h
42
43 # If you need a proxy to connect to the internet, uncomment the following line and comment the line below, a
44 # cmd.jvm.args=-Xms128m -Xmx1024m -Dapple.awt.UIElement=true -Dhttp.proxyHost=1.0.5.10 -Dhttp.proxyPort=8080
45 cmd.jvm.args=-Xms128m -Xmx1024m -Dapple.awt.UIElement=true
46
47 #-----
48 # The folder for the local package repository. By default, this folder is shared
49 # by all versions of Sencha Cmd. In other words, upgrading Sencha Cmd does not
50 # affect the local repository.
51
52 repo.local.dir=$(cmd.dir)/../repo
53
54 #-----
55 # By default, on 64-bit Windows, Sencha Cmd will prefer a 64-bit JRE over a
56 # 32-bit JRE if both are installed on the system. Restoring this option will
57 # disable the use of the 64-bit JRE and will instead only run a 32-bit JRE.
```

Search for the property definition of "cmd.jvm.args" (around line 45).

Comment the existing property definition and uncomment the line above it.

Then update the settings to match your proxy configuration.



The screenshot shows the 'sencha.cfg' file open in Notepad++. The file contains configuration settings for Sencha Cmd. Key sections include:

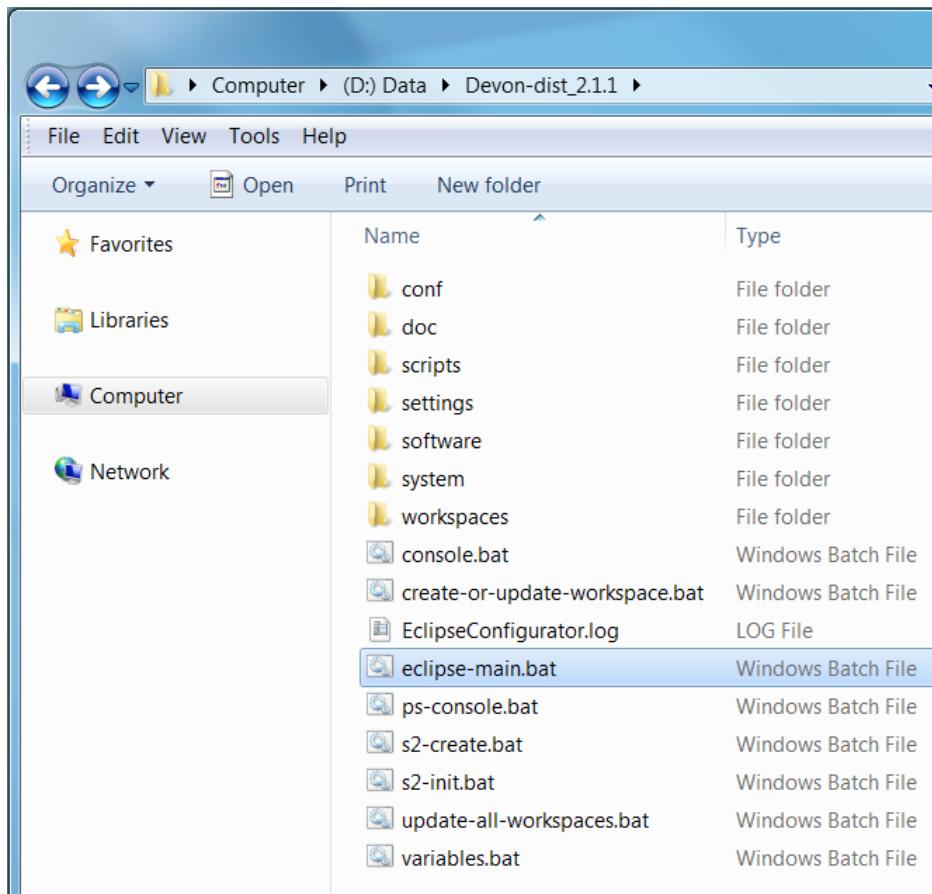
- cmd.minver** is set to 3.0.0.0.
- cmd.jvm.args** includes startup arguments for the JVM, such as `-Xms128m -Xmx2048m -Djava.awt.headless=true`.
- cmd.jvm.args** also includes proxy settings for HTTP and HTTPS, including `-Dhttp.proxyHost=1.0.5.10 -Dhttp.proxyPort=8080` and `-Dhttps.proxyHost=1.0.5.10 -Dhttps.proxyPort=8080`.
- repo.local.dir** is set to `$(cmd.dir)/../repo`.
- cmd.jvm.args** includes a note about preferring a 64-bit JRE over a 32-bit JRE on 64-bit Windows.

Notepad++ status bar details: Normal text file, length: 7419, lines: 175, Ln:44, Col:1, Sel:0|0, UNIX, UTF-8 w/o BOM, INS.

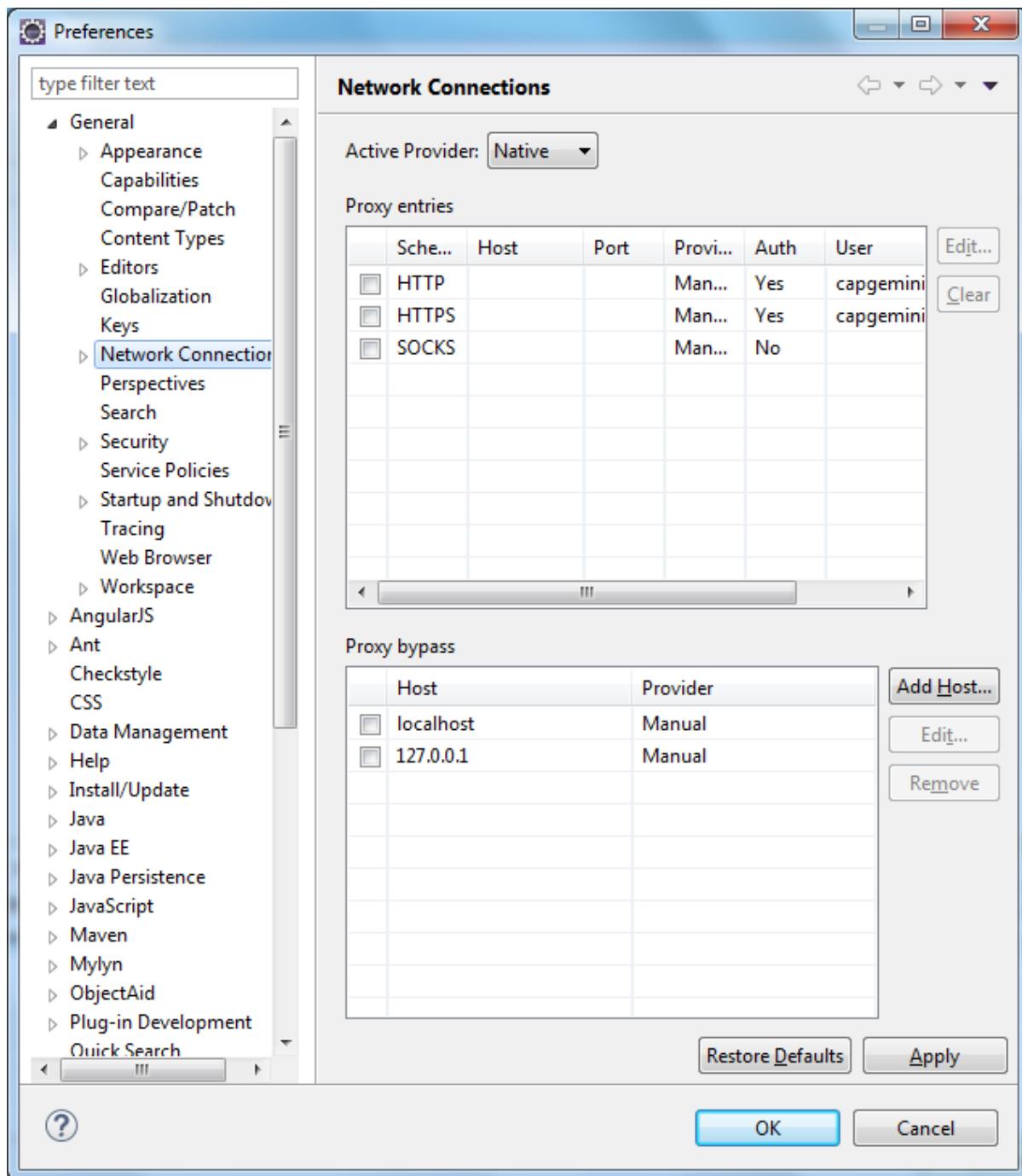
If your proxy does not require authentication, simply remove the "-Dhttp.proxyUser", "-DhttpProxyPassword", "-Dhttps.proxyUser" and "-Dhttps.proxyPassword" parameters.

## Eclipse

Open eclipse by executing "eclipse-main.bat".

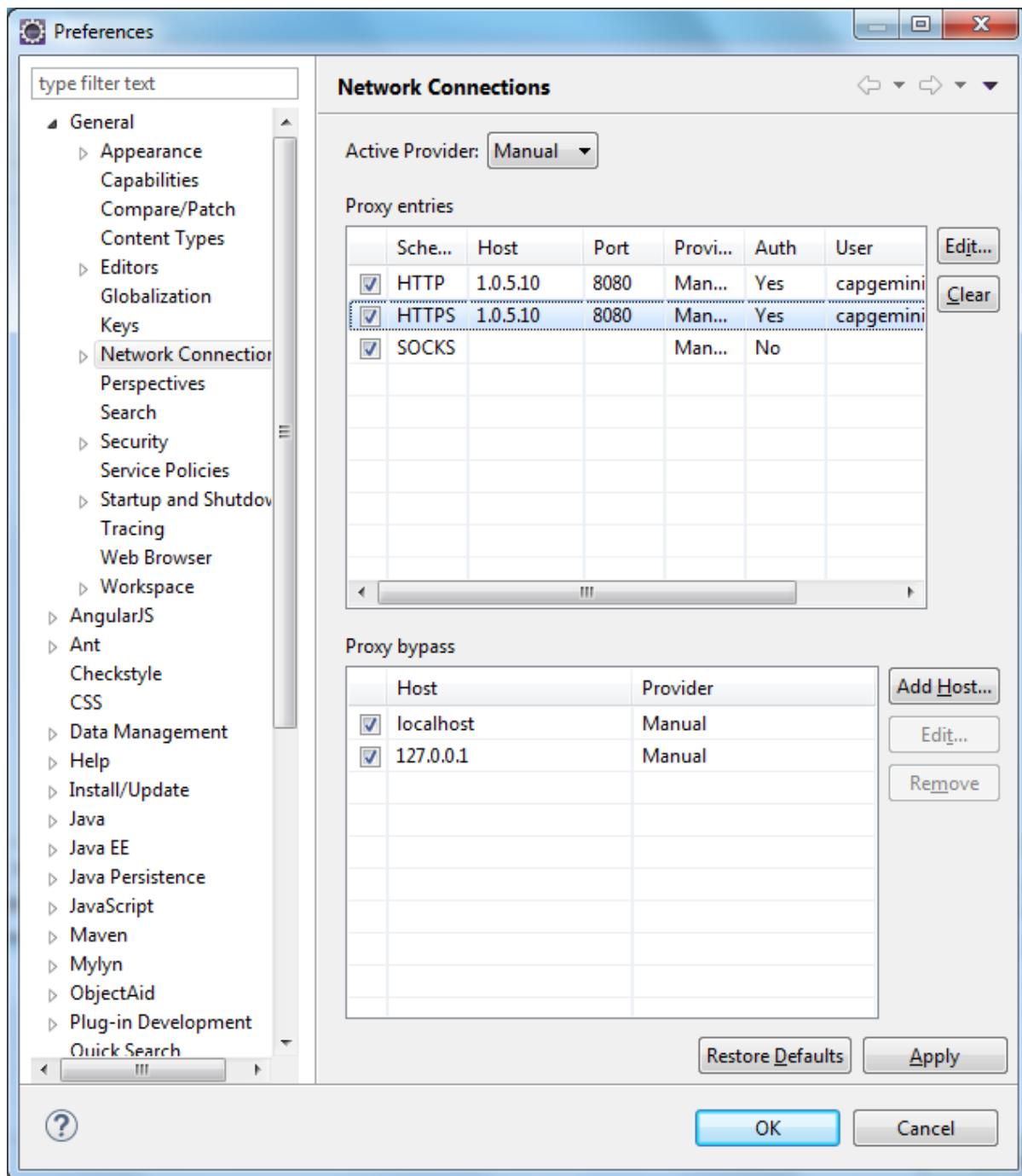


In the Eclipse preferences dialog, go to "General - Network Connection".



Switch from "Native" to "Manual"

Enter your proxy configuration



The working Devonfw environment is ready!

You can find example projects inside `workspaces\examples`. You can learn how to run the Devonfw example application [here](#).

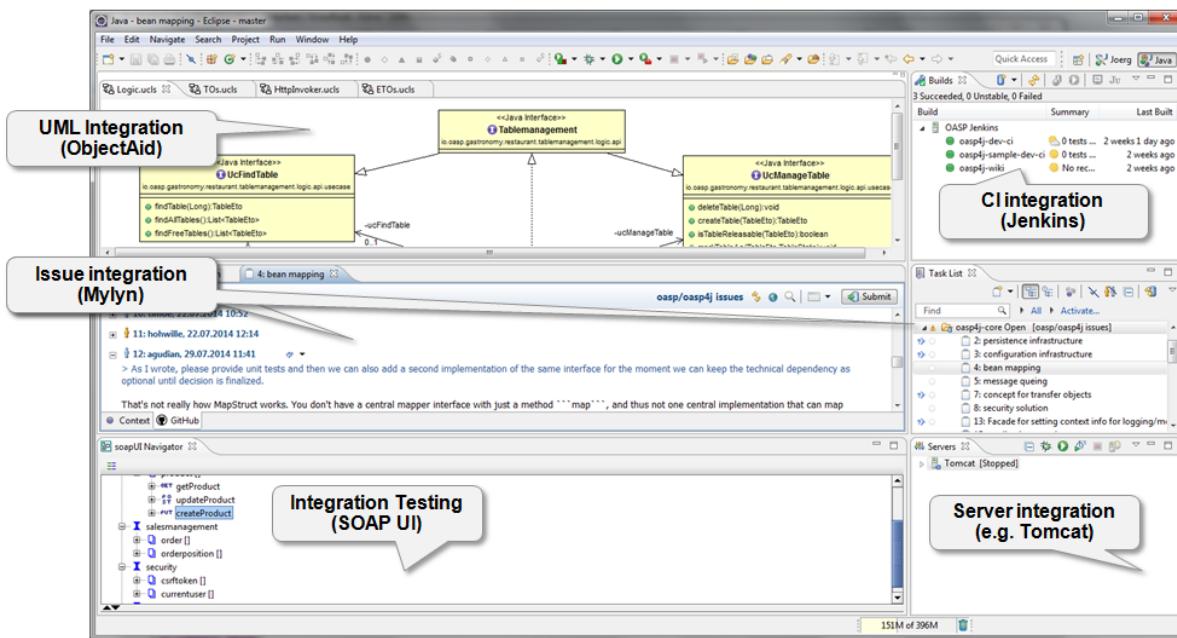
## 2.5 The Devon IDE

### 2.5.1 Introduction

The Devon IDE is the general name for two distinct versions of a customized Eclipse which comes in a Open Source variant, called OASP4J-IDE, and a more extended version included in the "Devon Dist" which is only available for Capgemini engagements.

[[getting-started-the-devon-ide\_features-and advantages]] === Features and advantages

Devonfw comes with a fully featured IDE in order to simplify the installation, configuration and maintenance of this instrumental part of the development environment. As it is being included in the distribution, the IDE is ready to be used and some specific configuration of certain plugins only takes a few minutes.



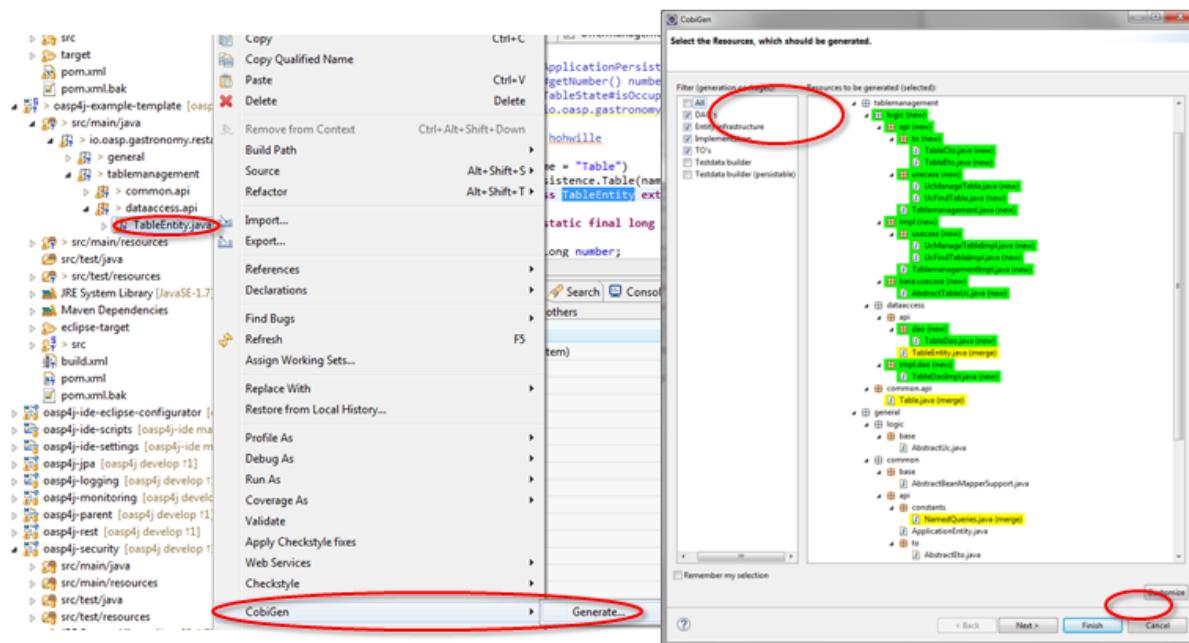
As with the remainder of the distribution, the advantage of this approach is that you can have as many instances of the -ide "installed" on your machine for different projects with different tools, tool versions and configurations. No physical installation and no tweaking of your operating system required. "Installations" of the Devon distribution do not interfere with each other nor with other installed software.

### Multiple Workspaces

There is inbuilt support for working with different workspaces on different branches. Create and update new workspaces with a few clicks. You can see the workspace name in the title-bar of your IDE so you do not get confused and work on the right branch.

### 2.5.2 Cobigen

In the Devon distribution we have a code generator to create CRUD code, called **Cobigen**. This is a generic incremental generator for end to end code generation tasks, mostly used in Java projects. Due to a template-based approach, Cobigen generates any set of text-based documents and document fragments.



Cobigen is distributed in the Devon distribution as an Eclipse plugin, and is available to all Devon developers for Capgemini engagements. Due to the importance of this component and the scope of its functionality, it is fully described [here](#).

### 2.5.3 IDE Plugins:

Since an application's code can greatly vary, and every program can be written in lots of ways without being semantically different, IDE comes with pre-installed and pre-configured plugins that use some kind of a probabilistic approach, usually based on pattern matching, to determine which pieces of code should be reviewed. These hints are a real time-saver, helping you to review incoming changes and prevent bugs from propagating into the released artifacts. Apart from Cobigen mentioned in the previous paragraph, the IDE provides CheckStyle, SonarQube, FindBugs and SOAP-UI. Details of each can be found in subsequent sections.

#### CheckStyle

[[getting-started-the-devon-ide\_what-is checkstyle]] ===== What is CheckStyle

[CheckStyle](#) is a Open Source development tool to help you ensure that your Java code adheres to a set of coding standards. Checkstyle does this by inspecting your Java source code and pointing out items that deviate from a defined set of coding rules.

With the Checkstyle IDE Plugin, your code is constantly inspected for coding standard deviations. Within the Eclipse workbench, you are immediately notified with the problems via the Eclipse Problems View and source code annotations similar to compiler errors or warnings. This ensures an extremely short feedback loop right at the developers fingertips.

[[getting-started-the-devon-ide\_why-use checkstyle]] ===== Why use CheckStyle

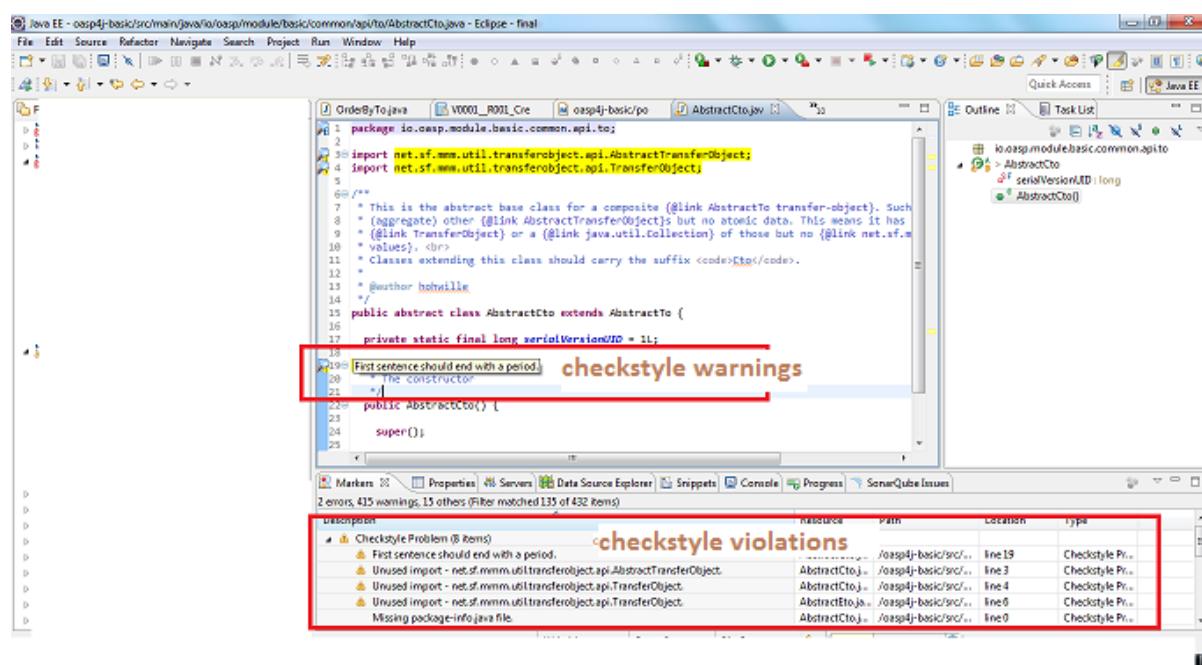
If your development team consists of more than one person, then obviously a common ground for coding standards (formatting rules, line lengths etc.) must be agreed upon - even if it is just for practical reasons to avoid superficial, format related merge conflicts. Checkstyle Plugin helps you define and easily apply those common rules.

The plugin uses a project builder to check your project files with Checkstyle. Assuming the IDE Auto-Build feature is enabled, each modification of a project file will immediately get checked by Checkstyle on file save - giving you immediate feedback about the changes you made. To use a simple analogy, the Checkstyle Plug-in works very much like a compiler but instead of producing .class files, it produces warnings where the code violates Checkstyle rules. The discovered deviations are accessible in the Eclipse Problems View, as code editor annotations and via additional Checkstyle violations views.

#### [[getting-started-the-devon-ide\_installation-of checkstyle]] ===== Installation of CheckStyle

After IDE installation, IDE provides default checkstyle configuration file which has certain check rules specified . The set of rules used to check the code is highly configurable. A Checkstyle configuration specifies which check rules are validated against the code and with which severity violations will be reported. Once defined a Checkstyle configuration can be used across multiple projects. The IDE comes with several pre-defined Checkstyle configurations. You can create custom configurations using the plugin's Checkstyle configuration editor or even use an existing Checkstyle configuration file from an external location.

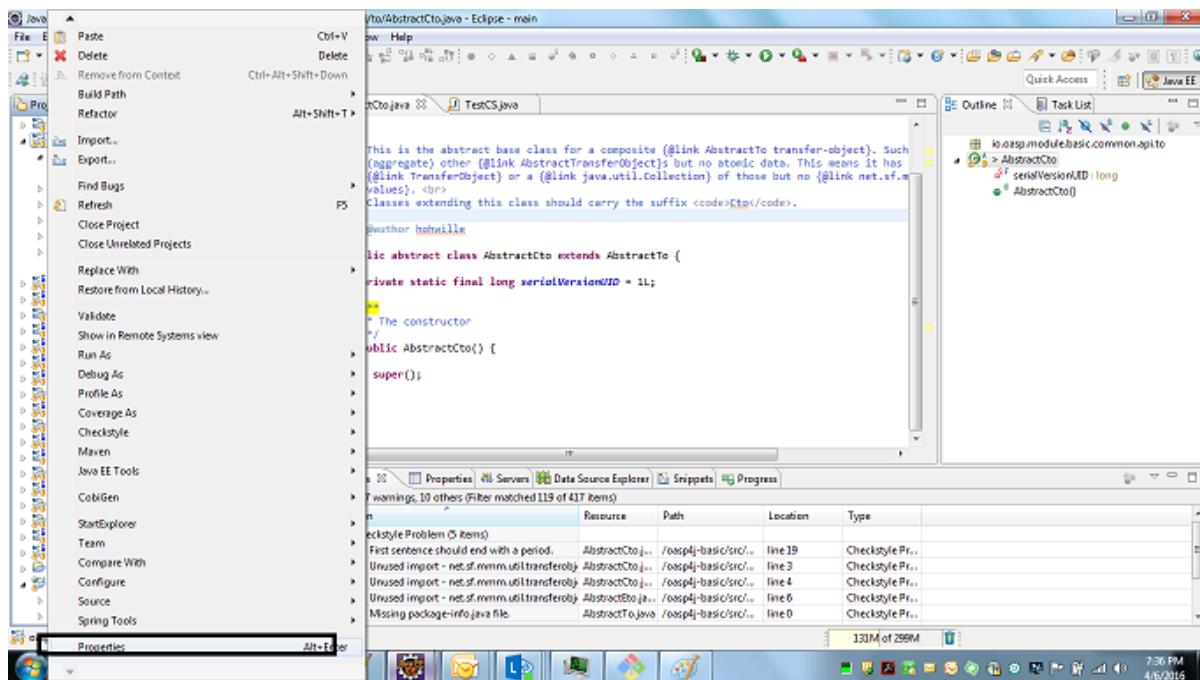
You can see violations in your workspace as shown in below figure.



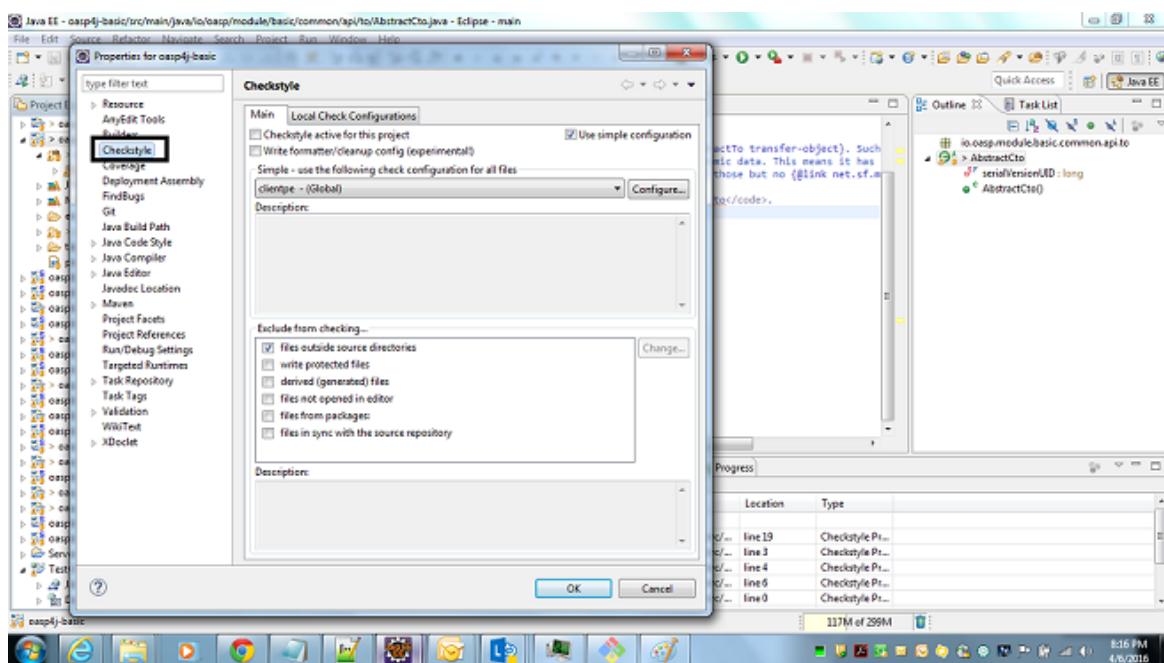
#### Usage

So, once projects are created, follow steps mentioned below, to activate checkstyle:

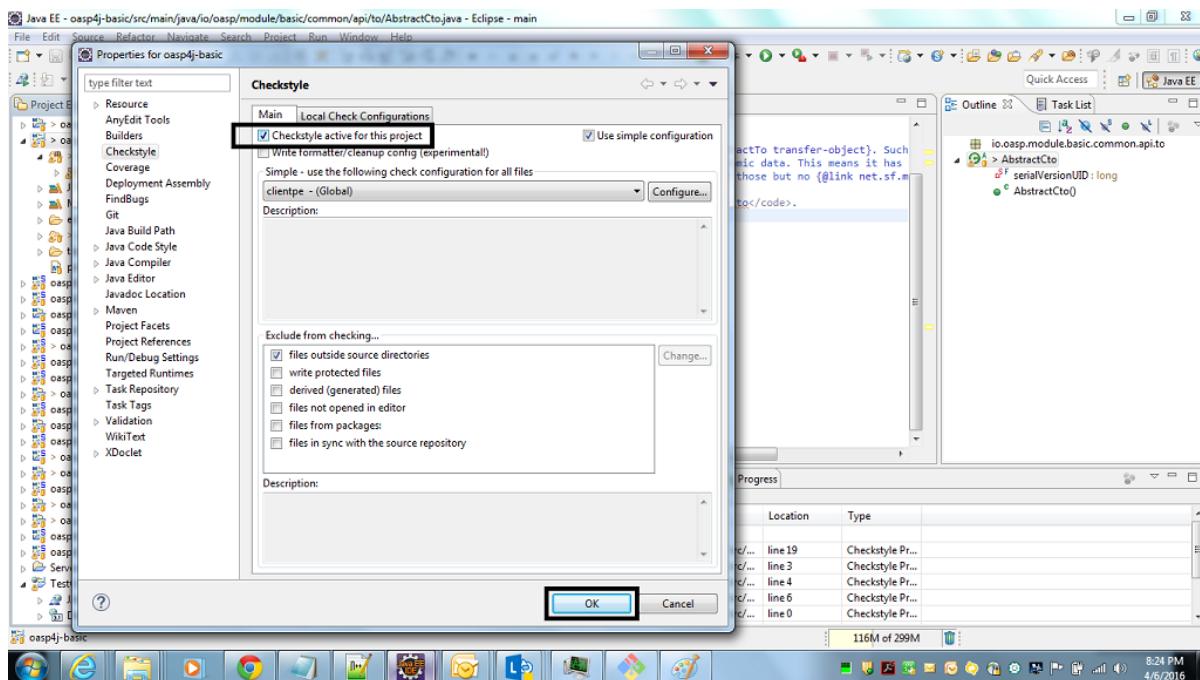
1. Open the properties of the project you want to get checked.



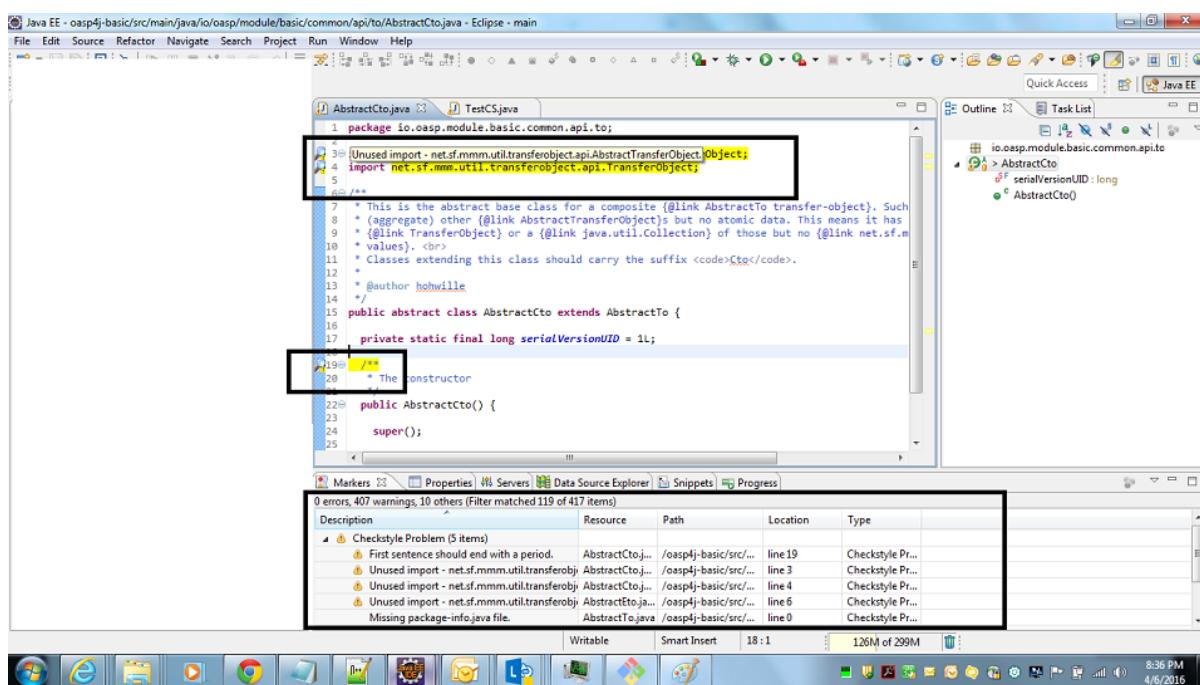
## 2. Select the Checkstyle section within the properties dialog .



## 3. Activate Checkstyle for your project by selecting the Checkstyle active for this project check box and press OK



Now Checkstyle should begin checking your code. This may take a while depending on how many source files your project contains. The Checkstyle Plug-in uses background jobs to do its work - so while Checkstyle audits your source files you should be able to continue your work. After Checkstyle has finished checking your code please look into your Eclipse Problems View. There should be some warnings from Checkstyle. These warnings point to the code locations where your code violates the preconfigured Checks configuration.



You can navigate to the problems in your code by double-clicking the problem in your problems view. On the left hand side of the editor an icon is shown for each line that contains a Checkstyle violation. Hovering with your mouse above this icon will show you the problem message. Also note the editor annotations - they are there to make it even easier to see where the problems are.

## FindBugs

[[getting-started-the-devon-ide\_what-is findbugs]] ===== What is FindBugs

[FindBugs](#) is an open source project for a static analysis of the Java bytecode to identify potential software bugs. Findbugs provides early feedback about potential errors in the code.

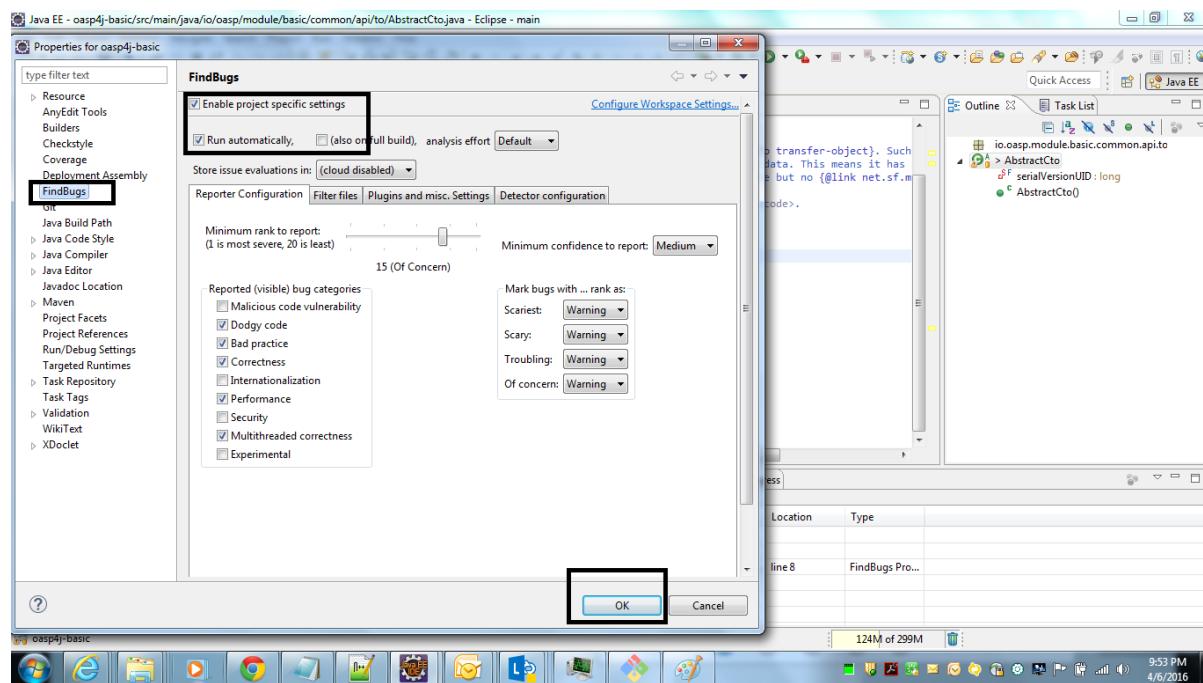
[[getting-started-the-devon-ide\_why-use findbugs]] ===== Why use FindBugs

It scans your code for bugs, breaking down the list of bugs in your code into a ranked list on a 20-point scale. The lower the number, the more hardcore the bug. This helps the developer to access these problems early in the development phase.

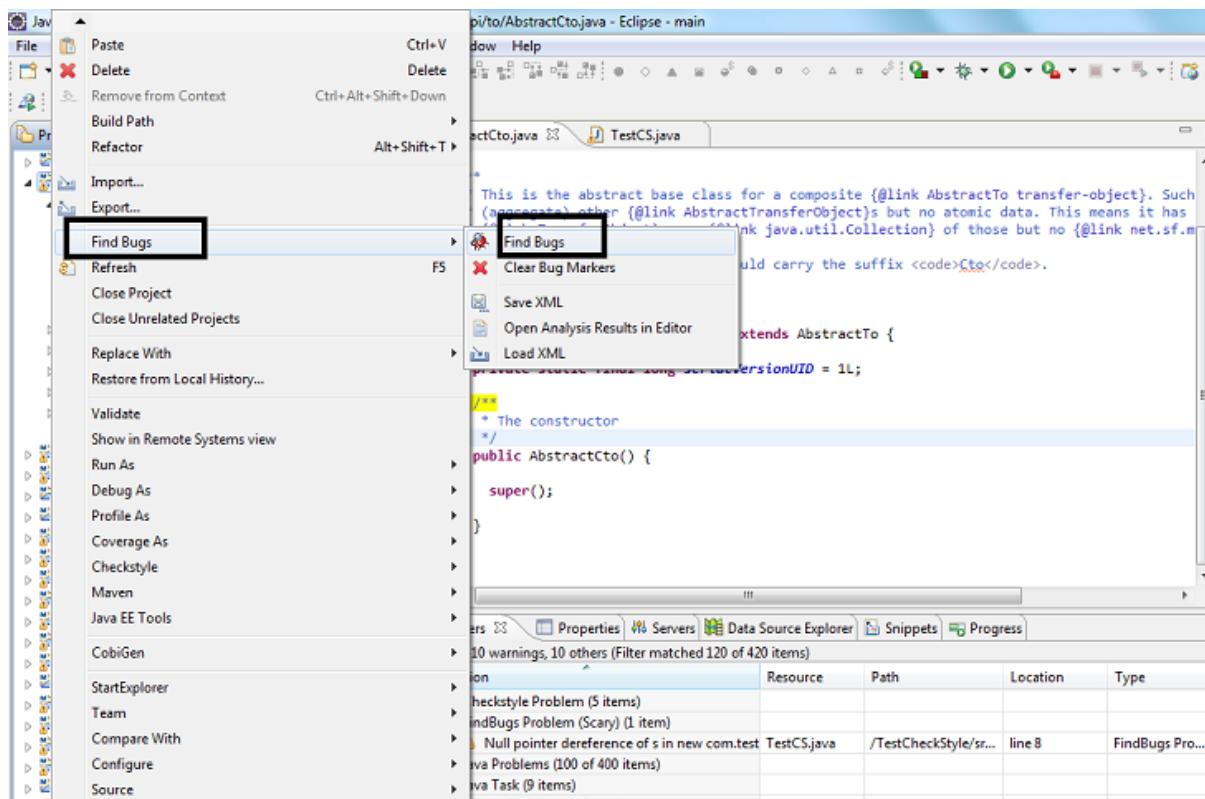
[[getting-started-the-devon-ide\_installation-and usage of findbugs]] ===== Installation and Usage of FindBugs

IDE comes preinstalled with FindBugs plugin.

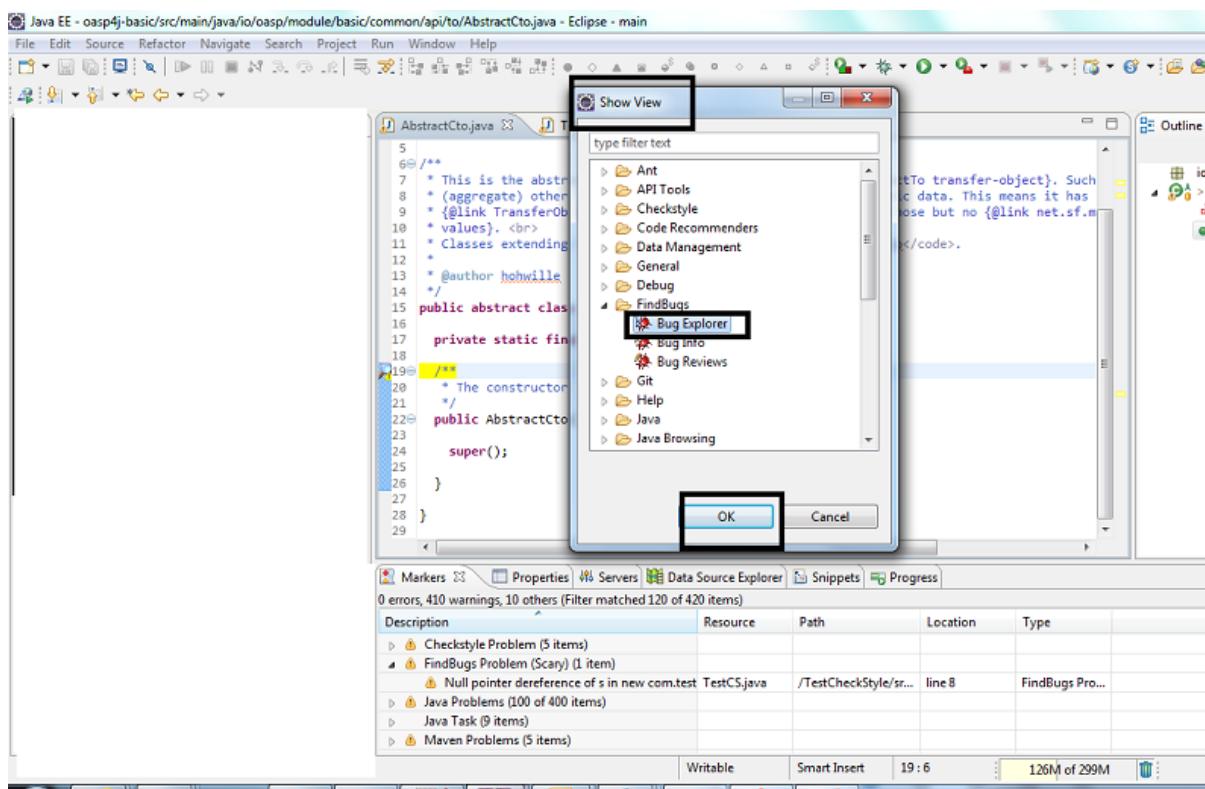
You can configure that FindBugs should run automatically for a selected project. For this right-click on a project and select Properties from the popup menu. via the project properties. Select FindBugs → Run automatically as shown below.

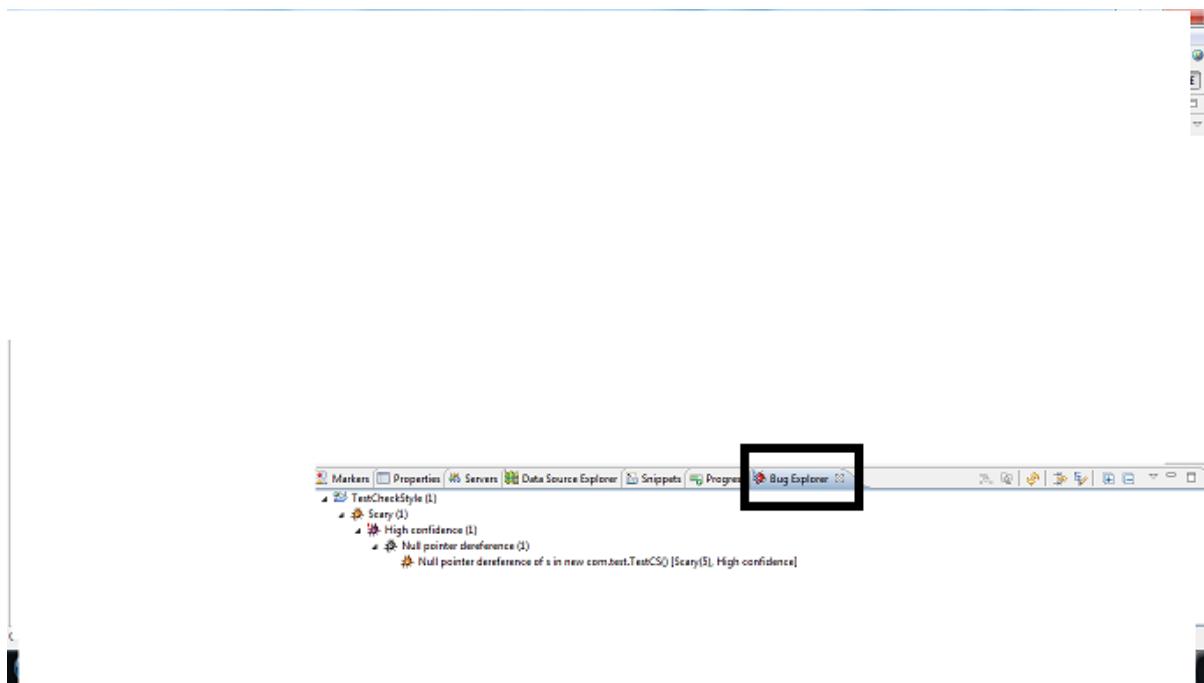


To run the error analysis of FindBugs on a project, right-click on it and select the Find Bugs... → Find Bugs menu entry.



Plugin provides specialized views to see the reported error messages. Select Window → Show View → Other... to access the views. The FindBugs error messages are also displayed in the Problems view or as decorators in the Package Explorer view.





## SonarLint

[[getting-started-the-devon-ide\_what-is sonarlint]] ===== what is SonarLint

[SonarLint](#) is an open platform to manage code quality. It provides on-the-fly feedback to developers on new bugs and quality issues injected into their code..

[[getting-started-the-devon-ide\_why-use sonarlint]] ===== Why use SonarLint

It covers seven aspects of code quality like junit, coding rules, comments, complexity, duplications, architecture and design and potential bugs. SonarLint has got a very efficient way of navigating, a balance between high-level view, dashboard and defect hunting tools. This enables to quickly uncover projects and / or components that are in analysis to establish action plans.

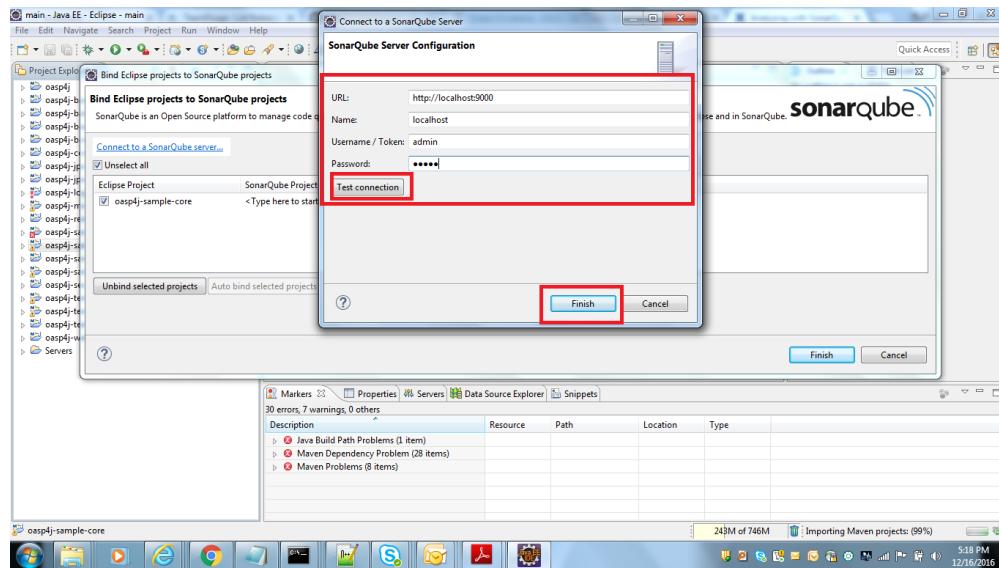
[[getting-started-the-devon-ide\_installation-and usage of sonarlint]] ===== Installation and usage of SonarLint

IDE comes preinstalled with SonarLint. To configure it , please follow below steps:

First of all, you need to start sonar service. For that , go to software folder which is extracted from Devon-dist zip, choose sonarqube#bin#<choose appropriate folder according to your OS>-#and execute startSonar bat file.

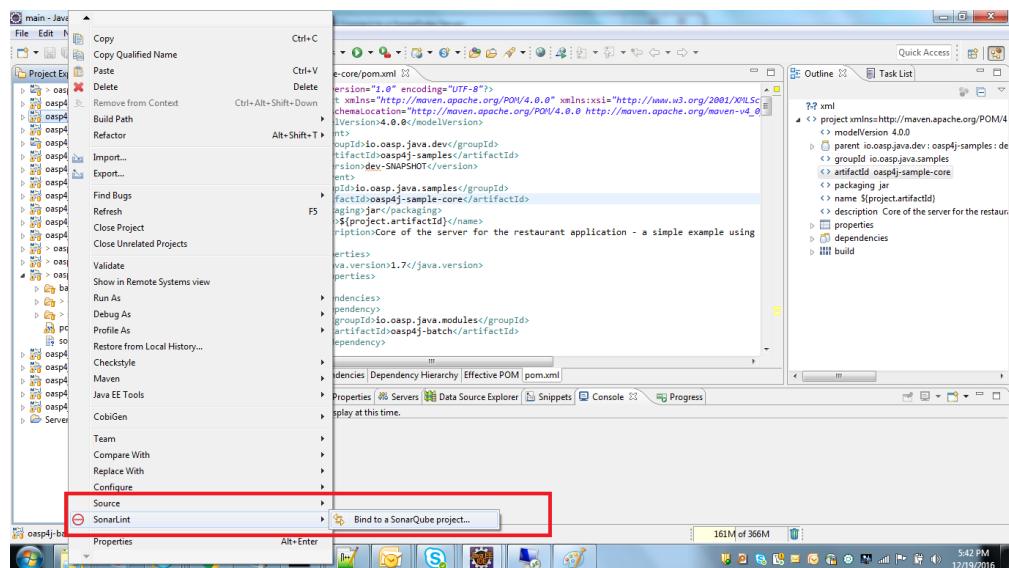
If your project is not already under analysis, you'll need to declare it through the SonarQube web interface as described [here](#). Once your project exists in SonarQube, you're ready to get started with SonarQube in Eclipse.

SonarLint in Eclipse is pre-configured to access a local SonarQube server listening on <http://localhost:9000/>. You can edit this server, delete it or add new ones. By default, user and password is "admin". If sonar service is started properly, test connection will give you successful result.

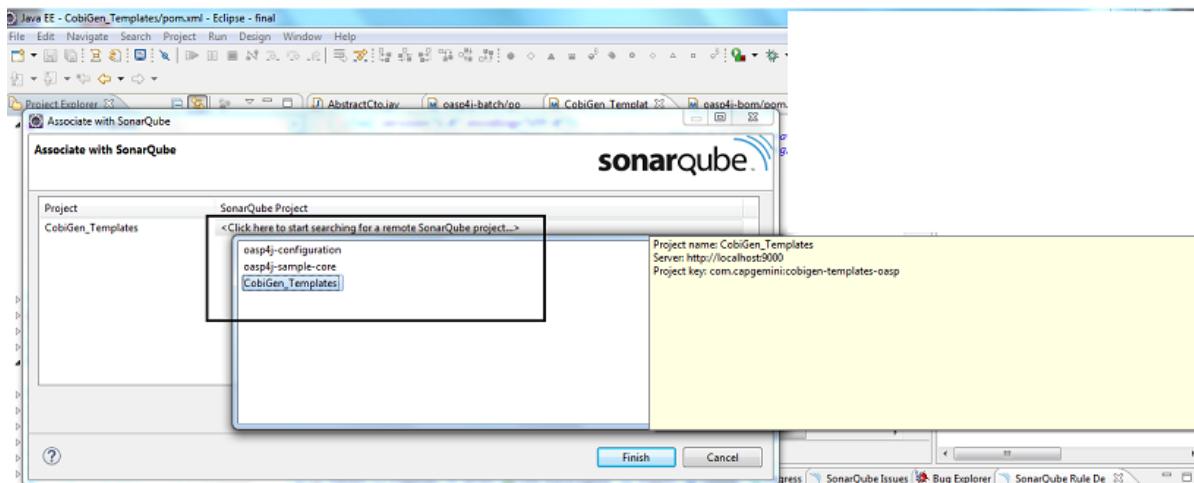


For getting a project analysed on sonar, refer this <http://docs.sonarqube.org/display/SONAR/Analyzing+Source+Code> [link].

Linking a project to one analysed on sonar server.



In the SonarQube project text field, start typing the name of the project and select it in the list box:

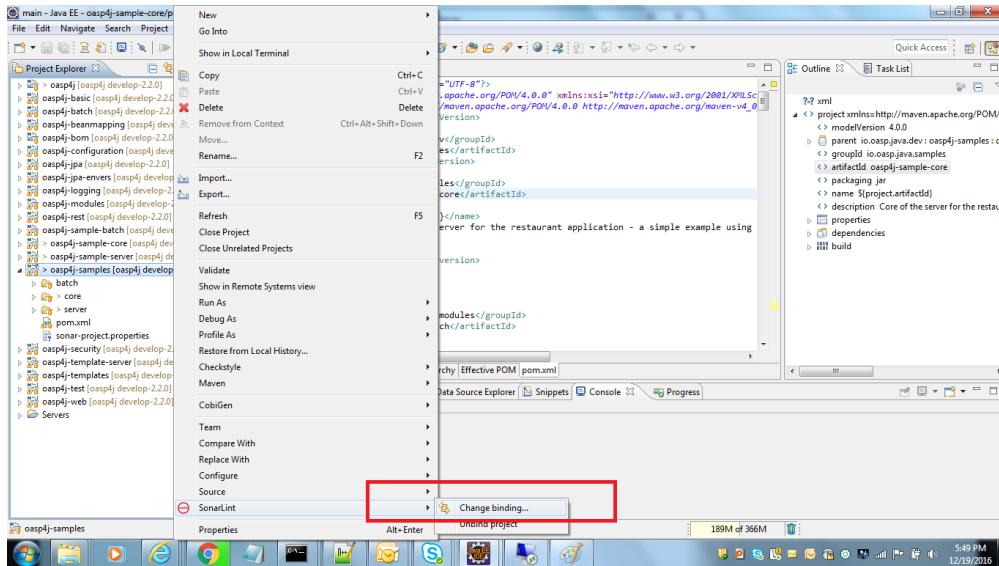


Click on Finish. Your project is now associated to one analyzed on your SonarQube server.

## Changing Binding

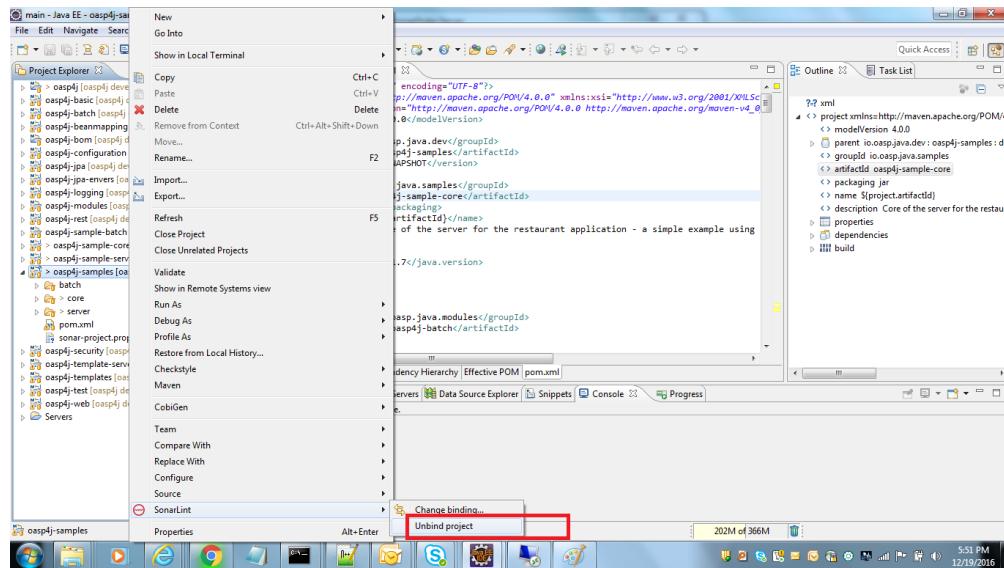
At any time, it is possible to change the project association.

To do so, right-click on the project in the Project Explorer, and then SonarQube > Change Project Association.



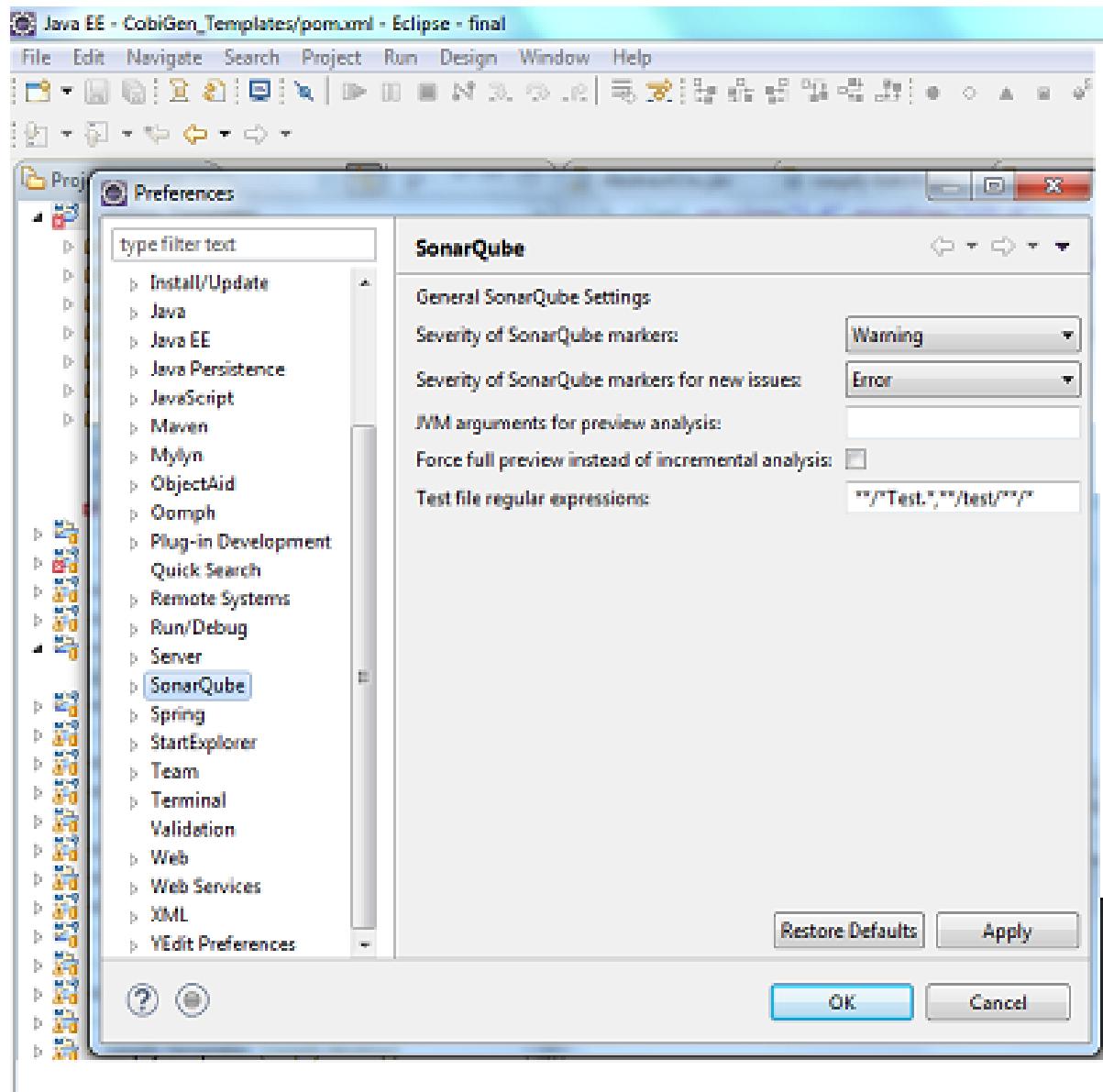
## Unbinding a Project

To do so, right-click on the project in the Project Explorer, and then SonarQube > Remove SonarQube Nature.

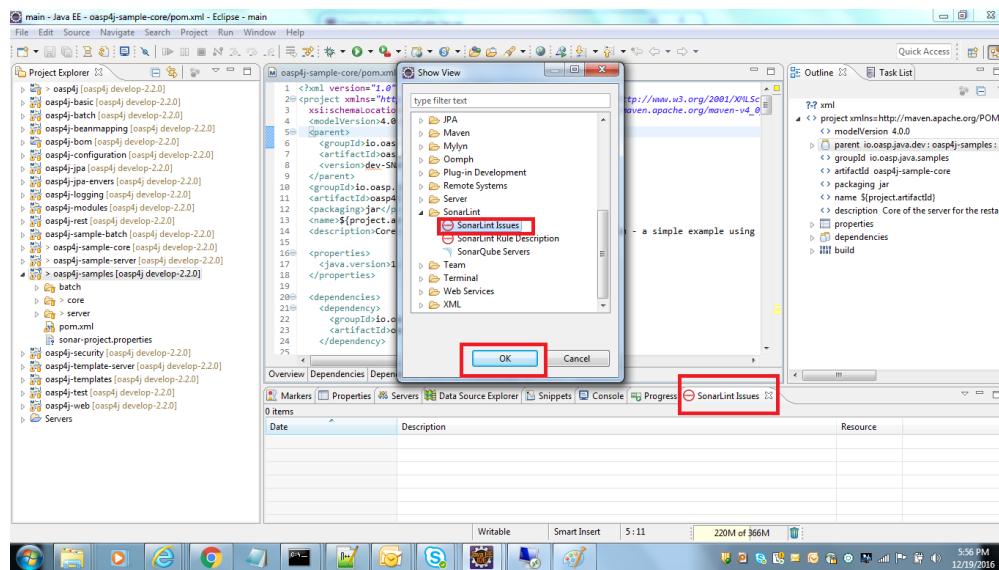


## Advanced Configuration

Additional settings (such as markers for new issues) are available through Window > Preferences > SonarLint



To look for sonarqube analysed issue, go to Window#Show View# Others#SonarLint#SonarLint Issues. Now you can see issues in soanrqube issues tab as shown



Or you can go to link <http://localhost:9000> and login with admin as id and admin as password and goto Dashboard.you can see all the statistics of analysis of the configured projects on sonar server.

## 2.6 Sample Application

Devonfw comes with a sample application, which showcases the most common features of a Devonfw application and its proposed best practices. The sample application also serves as a final test to make sure that your environment is setup correctly.

This section explains how to download the sample application and run it. Setup Devonfw environment and follow the below steps:

[[getting-started-running-sample-application\_downloading-sample application]] == Downloading Sample Application

First of all, download or clone the application from the [GitHub repository](#). We recommend to use the Git Bash to work with our repository of GitHub. You can download Git and Git Bash for Windows system [here](#). Also you can learn more about the best practices of work with Git and GitHub on Devonfw [here](#).

To clone the repository, open the Git Bash console and run the following command:

```
git clone https://github.com/devonfw/devon.git
```

Now, let's discuss how to run the Sample Application.

[[getting-started-running-sample-application\_running-sample application]] == Running Sample Application



### Using devcon

You can automate the following steps using devcon with the `devon oasp4j run` command [learn more here](#)

We can run the application in two ways or whatever other Devonfw application based on Spring boot: [Running with the embedded Tomcat 8 server of Spring Boot](#) or [using an external Tomcat 8 server](#).

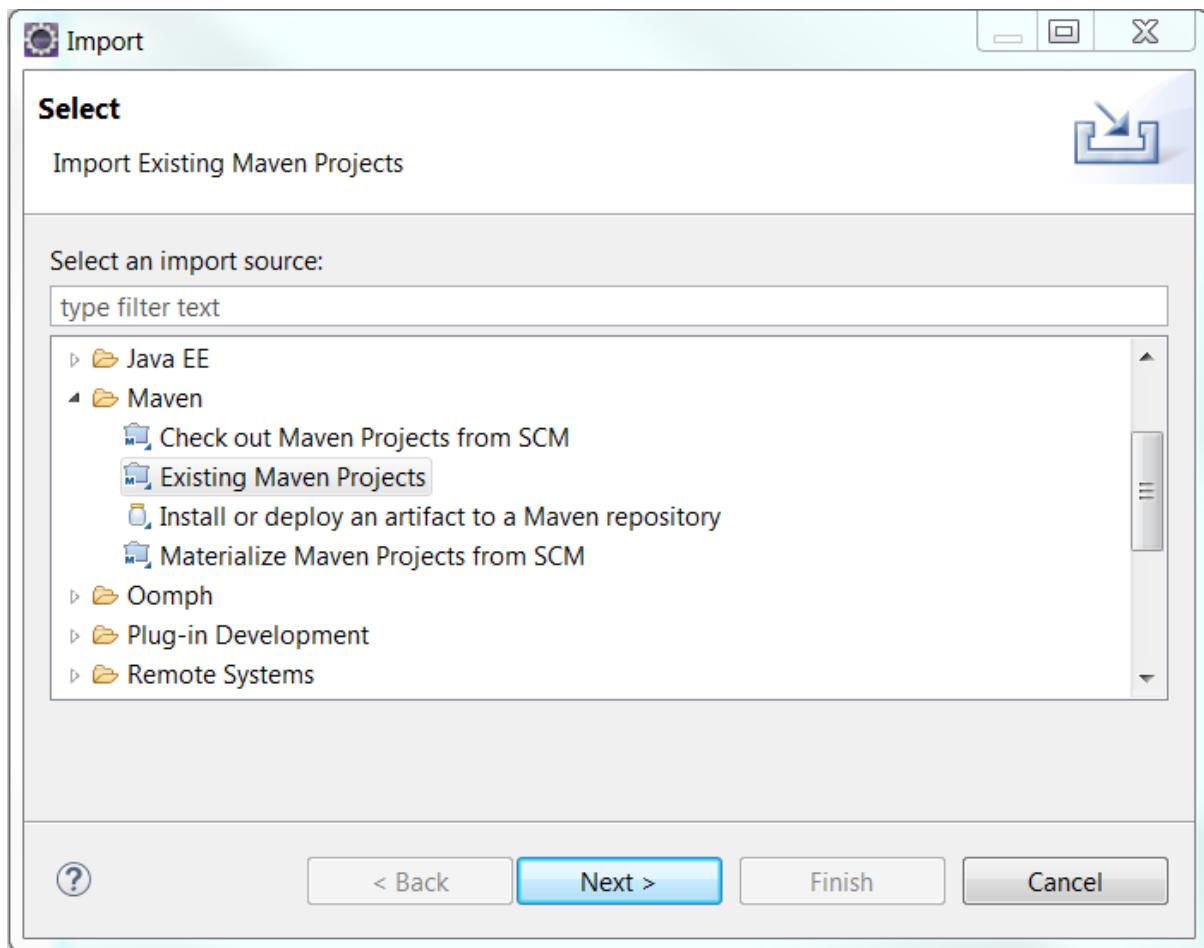
[[getting-started-running-sample-application\_embedded-tomcat server]] === Embedded Tomcat server

All Spring boot application have their own embedded Tomcat server. This feature help us to deploy the application in the develop time without the need to create an server. To run the application with this mode, you need to open Eclipse of your Devonfw distribution and perform the following steps:

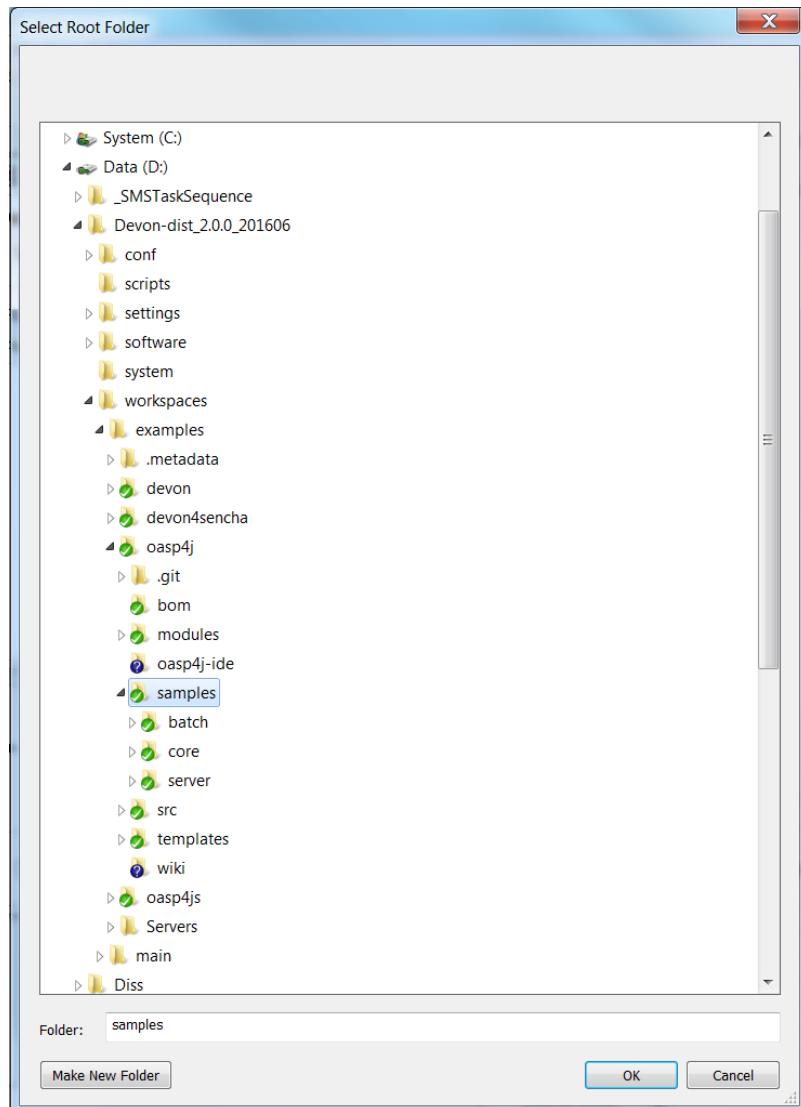
[[getting-started-running-sample-application\_step-1: import the project]] ===== Step 1: Import the project

First of all, import above Sample Application into Eclipse using the following steps:

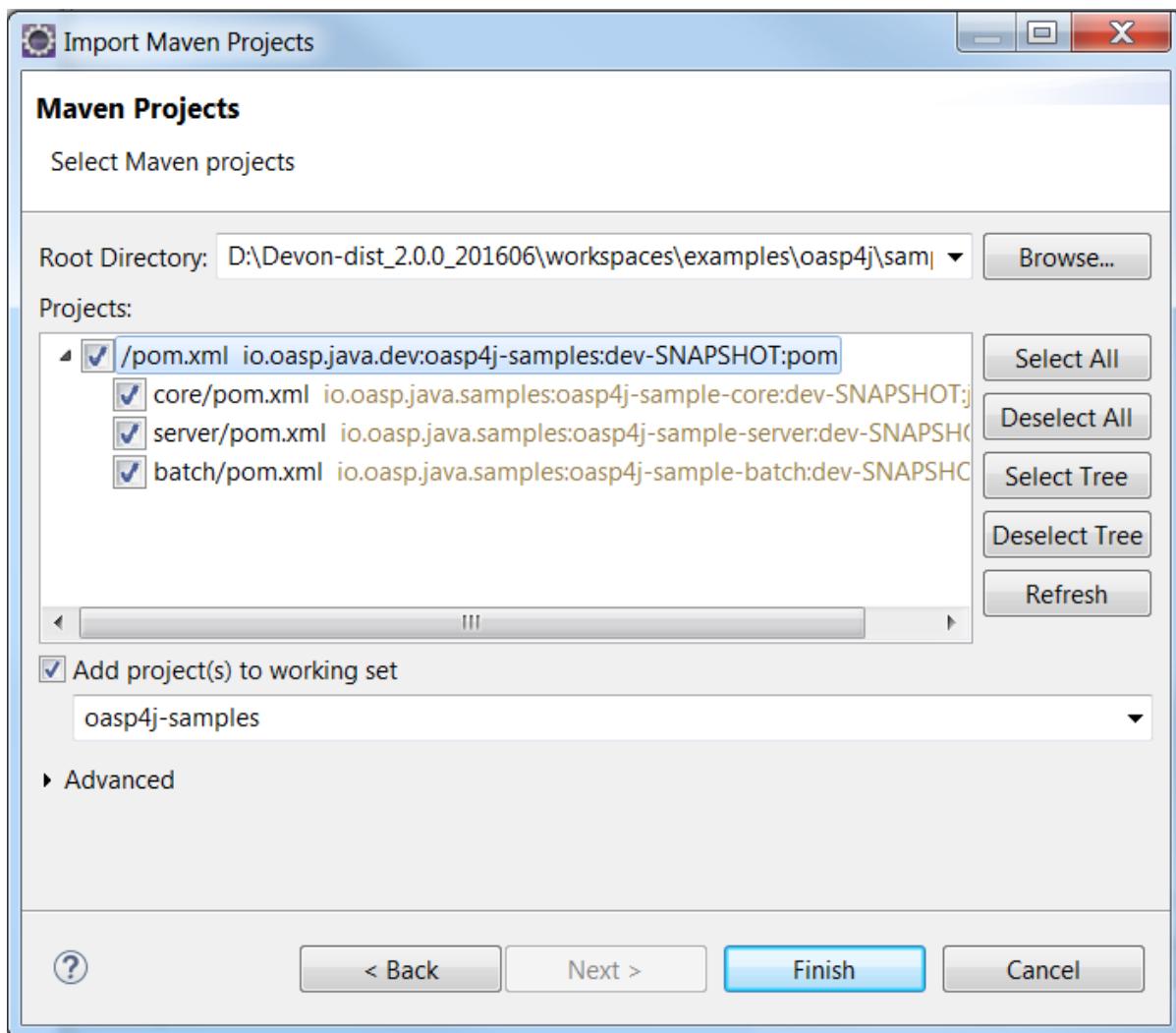
1. Open Eclipse by executing "eclipse-main.bat"
2. Select "File - Import"
3. Select "Maven - Existing Maven Projects"



4. Select the directory "workspaces/examples/oasp4j/samples"



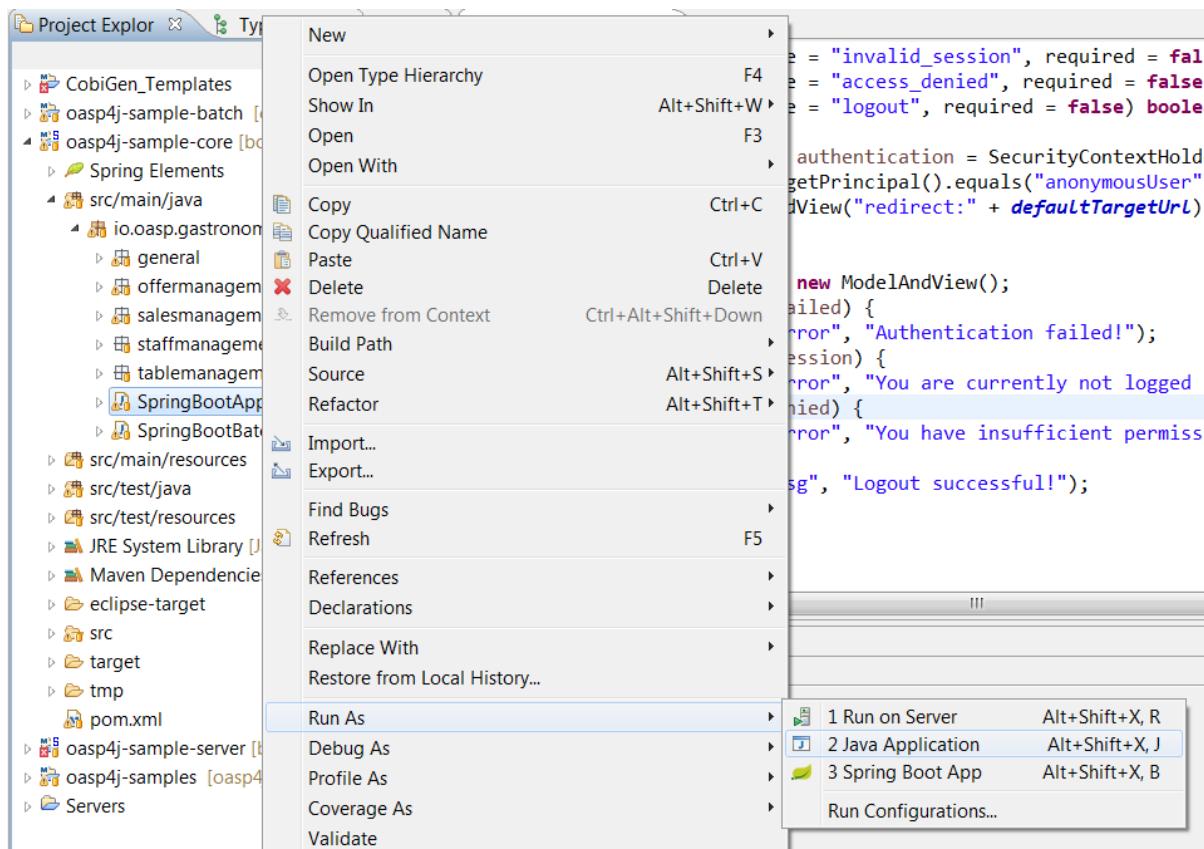
5. Press "Finish"



6. Wait for Eclipse to finish importing the sample project. This process might take several minutes, depending on the speed of your internet connection.

[[getting-started-running-sample-application\_step-2: run the application]] ===== Step 2: Run the application

The embedded Tomcat server of the Spring Boot Application can be running with a simple right click on the Java file **oasp4j-sample-core/src/main/java/io/oasp/gastronomy/restaurant/SpringBootApp.java** > Run as... > 2 Java Application



Now you can open your favorite browser and see the Sample Application running in the next URL

```
http://localhost:8081/oasp4j-sample-server/
```

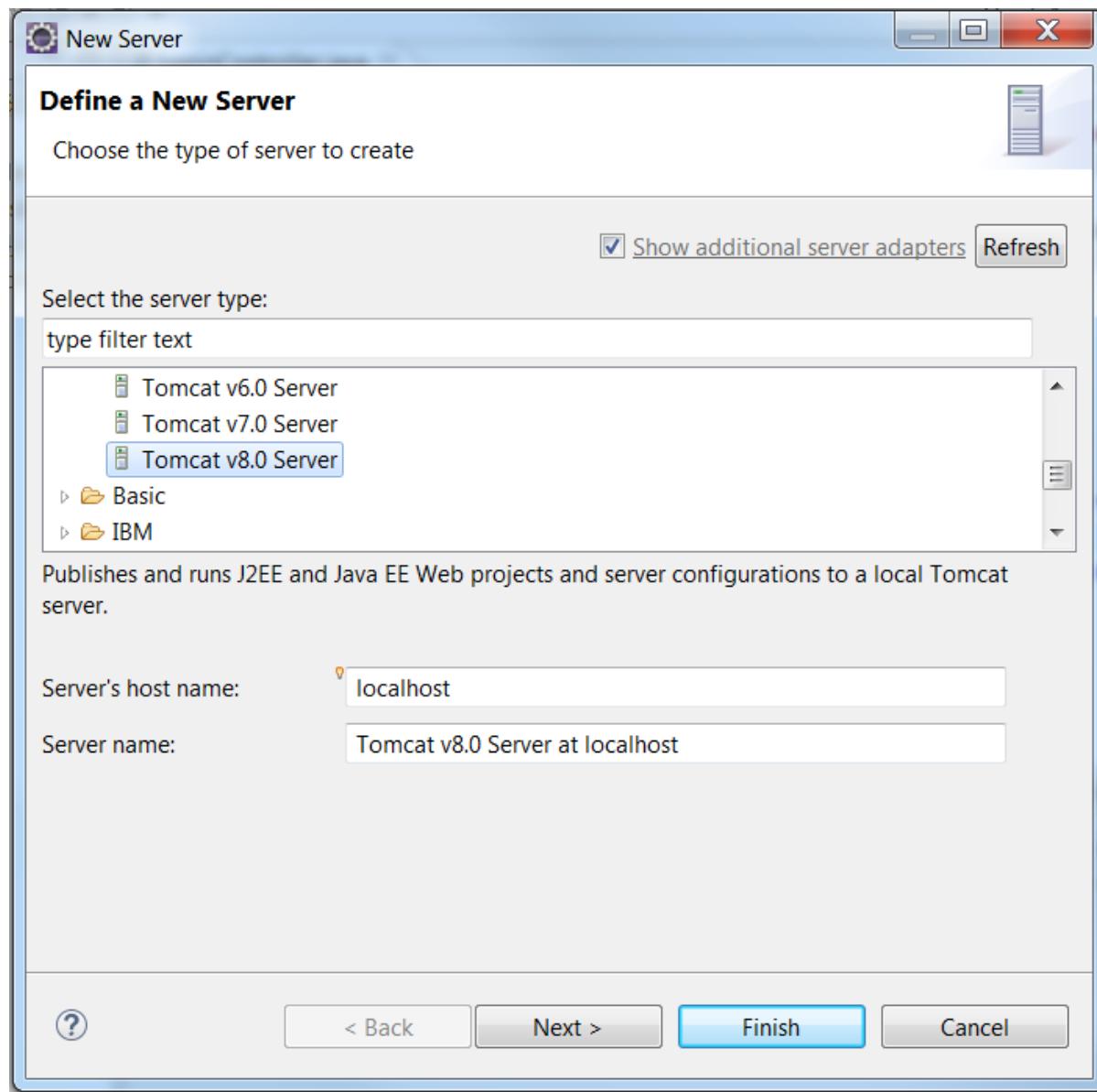
As you can see, this embedded server is a very comfortable way to start the application. When you run the application with this server, you need to take into account that the application is using the properties of the *application.properties* file in the **src/main/resources/config** folder.

[[getting-started-running-sample-application\_external-tomcat server]] === External Tomcat server

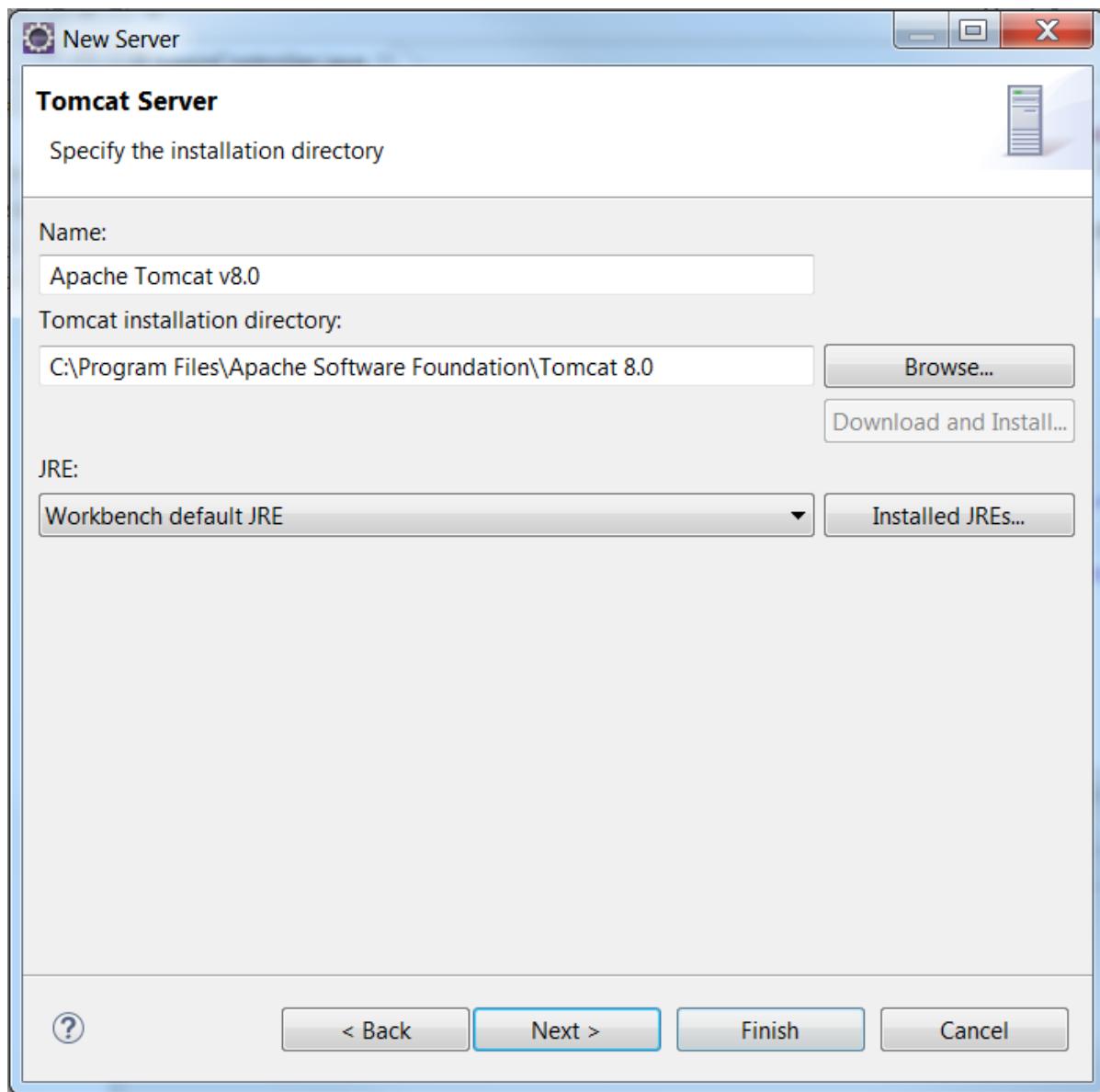
Normally we want to deploy our application in an external server, so we can do this on Eclipse, creating a new server. The steps to run an OASP4J application in an external Tomcat server are the next:

[[getting-started-running-sample-application\_step-1: create new tomcat server]] ===== Step 1: Create new Tomcat server

- Open the Server view in Eclipse if you don't have it opened (main menu → Windows → Show View → Servers).
- In servers view do right click → New → Server



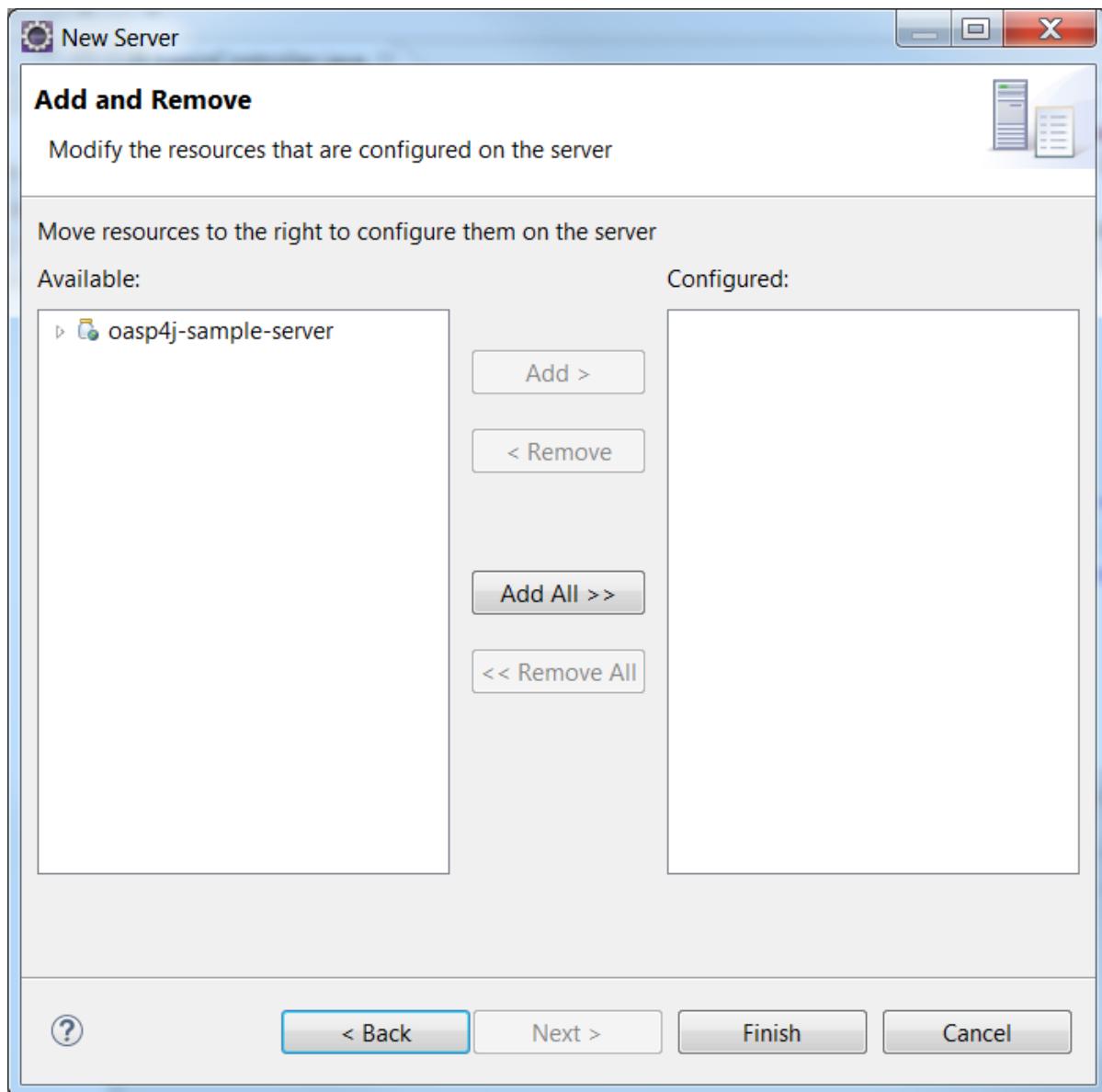
- In the wizard panel you need to configure the server as Tomcat 8 and press Next.



It's possible that Eclipse ask you for the path of Tomcat 8. If you don't have Tomcat 8 in your distribution of Devonfw you can download it [here](#) and extract the zip in the software folder of your distribution.

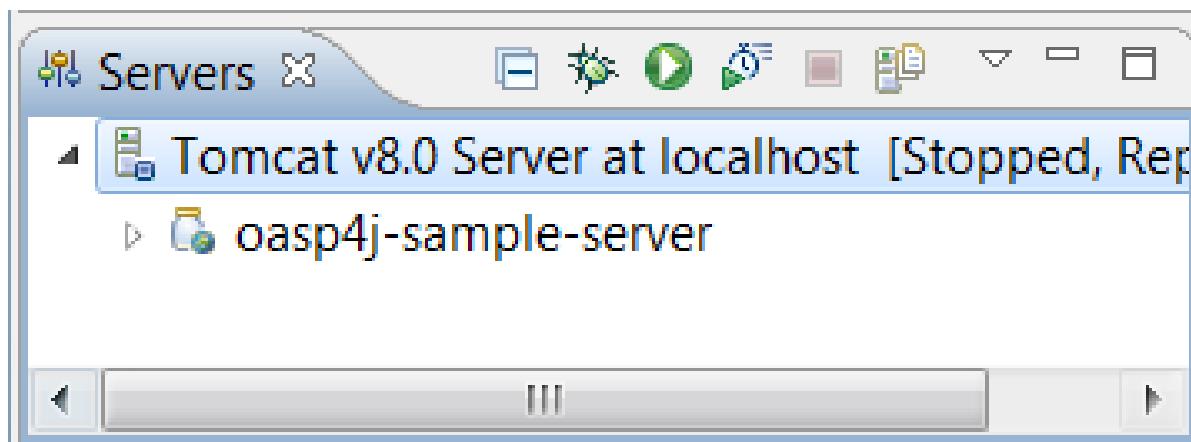
[[getting-started-running-sample-application\_step-2: add the application in the server]] ===== Step 2:  
Add the application in the server

Now we need to go to the Server View and do right click in your new Tomcat 8 server → Add and Remove... → add the oasp4j-sample-server → Finish.



[[getting-started-running-sample-application\_step-3: run the server]] ===== Step 3: Run the server

Now we need to press Run or Debug in the Server View with our new Tomcat server selected. And the application will be deployed by the server.



Now you can open your favorite browser and see the Sample Application running in the next URL

```
http://localhost:8080/oasp4j-sample-server/
```

As you can see the port is 8080, but you can modify this properties in the Tomcat server properties (double click over the server in the *Servers* view) or in **src/main/resources/application.properties**. You need to take into account that these are the properties that the application will use if you deploy the application in the context of an external Tomcat server.

[[getting-started-running-sample-application\_step-4: login at the application]] ===== Step 4: Login at the application

When everything is already successfully set up, we can finally enter in the application by using default user credentials:

Username	Password
chief	chief
waiter	waiter
cook	cook

Depending on which user/pass is chosen to login, the application will behave differently. For instance, by using **chief/chief** will grant access to both the table management and the kitchen management. It will not happen by using the other two credentials, **waiter/waiter** and **cook/cook** which will grant access just to the table management or to the kitchen management respectively.

## 2.7 Devonfw Distribution Structure

In this section, you will find outlook on the Devonfw distribution structure that you will find right after downloading the zip. In short, the use of each file and folder will be explained here.

Therefore, after unzipping the Devonfw distribution, you will find the following directory structure as shown in the image:

Name	Type
doc	File folder
scripts	File folder
settings	File folder
software	File folder
system	File folder
workspaces	File folder
console.bat	Windows Batch File
create-or-update-workspace.bat	Windows Batch File
eclipse-main.bat	Windows Batch File
ps-console.bat	Windows Batch File
s2-create.bat	Windows Batch File
s2-init.bat	Windows Batch File
update-all-workspaces.bat	Windows Batch File
variables.bat	Windows Batch File

[[getting-started-distribution-structure\_understanding-the structure]] == Understanding the structure

In the above image, you can find different folders and executable `.bat` files. You will find below the use of `create-or-update-workspace.bat` and `update-all-workspaces.bat` files. These are the scripts which you will execute to obtain the whole Devonfw structure:

- The **create-or-update-workspace.bat** file will create the `conf` directory that stores the Maven local repository and two configuration files (the `settings.json` with distribution information and the `settings.xml` with the Maven connection settings). Alongside with that it will create the `eclipse-main.bat` to start Eclipse.
- The **update-all-workspaces.bat** file sets some Eclipse preferences for workspaces and creates all the Eclipse `.bat` launchers related to the projects in the `workspaces` directory.

Hence, after executing these two scripts, following structure will be generated:

Name	Type
conf	File folder
doc	File folder
scripts	File folder
settings	File folder
software	File folder
system	File folder
workspaces	File folder
console.bat	Windows Batch File
create-or-update-workspace.bat	Windows Batch File
EclipseConfigurator.log	LOG File
eclipse-examples.bat	Windows Batch File
eclipse-main.bat	Windows Batch File
ps-console.bat	Windows Batch File
s2-create.bat	Windows Batch File
s2-init.bat	Windows Batch File
update-all-workspaces.bat	Windows Batch File
variables.bat	Windows Batch File

Thereon, You will have the new *conf* directory as expected and the *eclipse-main.bat* and the *eclipse-examples.bat* related to the *main* and *examples* directories within the *workspaces* folder.

This is the final structure. A detailed explanation for each file and folder is given below:

### 2.7.1 conf

As mentioned previously, this directory is generated by executing the *create-or-update-workspace.bat*. This is the directory where Maven will store its local repository, in a *.m2/repository* path. Moreover, in the *conf* directory, you can find two settings files. The *settings.json* with distribution information and the *settings.xml* with the Maven connection settings.

### 2.7.2 doc

Here, you can find, the documentation related to both, the starting development tasks with the framework and implementing its more advanced features in pdf format.

### 2.7.3 scripts

This folder stores the scripts referenced in the *.bat* files in the root directory. These scripts are related to internal tasks of the distribution.

### 2.7.4 settings

This directory stores the elements required for internal functionality of the distribution. Here, you can find the configuration files of different software, included in the distribution, such as Eclipse, Maven, Sonarqube and several more.

## 2.7.5 software

All the software resources that the distribution needs are stored in this folder. Internally, the distribution will search here for available software. Therefore, all the programs, plugins and tools needed by the distribution must be located in this directory.

## 2.7.6 system

This is another directory with internal elements. In *system* folder, you can find some files related to the environment configuration.

## 2.7.7 workspaces

This is the directory to store all the projects. One must keep in mind that the content of this folder will be associated with a Eclipse *.bat* launcher files through the *update-all-worksapces.bat* script. So if you want the separated Eclipse instances for two different projects, you must declare these projects in separate directories within the *workspaces* folder.

To conclude, if you have a *workspaces/project01* and a *workspaces/project02* projects, then the *update-all-workspaces.bat* script will create a *eclipse-project01.bat* launcher and a *eclipse-project02.bat* launcher in the root folder of the distribution. Thus, you can have access to the different Eclipse instances with different configurations for each project.



### Using devcon

You can automate this operation using devcon with the `devon workspace create` command [learn more here](#)

## 2.7.8 console.bat

This script launches the distribution's *cmd*. Meaning, within this *cmd*, you have access to the software located in the *software* folder, so that you can use the tools "installed" in that folder although you don't have this installed on your machine. Therefore, it is important to always run this *cmd* (launching the *console.bat* script) to make use of the software related to the distribution.

## 2.7.9 create-or-update-workspace.bat

This script is already explained [at the beginning of this chapter](#).

## 2.7.10 EclipseConfigurator.log

This is a file for internal usage and records the logs of the *create-or-update-workspace.bat* and the *update-all-workspaces.bat* scripts.

## 2.7.11 eclipse-project.bat

These files are used to have different Eclipse instances related to the different projects located into the *workspaces* directory. Therefore, for each project in the *workspaces* directory, the *update-all-*

---

*workspaces.bat* script will create an Eclipse launcher with structure *eclipse-< projectName >.bat*. In such a way, you can have different Eclipse environments with different configurations related to the different projects of the *workspace* directory.

[[getting-started-distribution-structure\_s2-create.bat-and s2-init.bat]] === s2-create.bat and s2-init.bat These scripts relate to the *Shared Services* functionality included in Devonfw. The *s2-init.bat* configures the *settings.xml* file to connect to an Artifactory Repository. The *s2.create.bat* generates a new project in the *workspaces* directory and does a checkout of a Subversion repository inside. Each script needs to be launched from the distribution's cmd (launching the *console.bat* script) and some parameters to work properly.

## 2.7.12 update-all-workspaces.bat

This script is already explained [at the beginning of this chapter](#).

## 2.7.13 variables.bat

This script is related to the internal functionality of the distribution. The script stores some variables that are used internally by the distribution scripts.

## 2.8 Create New Application



### Using Devcon

You can automate the following steps using Devcon with the `devon oasp4j create` command. To learn more, click [here](#).

[[getting-started-creating-new-devonfw-application\_running-the archetype]] == Running the archetype

In order to create a new application, you must use the archetype provided by devon which uses the maven archetype functionality.

There are two alternatives for using the archetype to create a new application. One is to create using command line. Another way is within eclipse, which is a more visual manner.

[[getting-started-creating-new-devonfw-application\_using-command line]] === Using command Line

[[getting-started-creating-new-devonfw-application\_step-1: open the console]] ===== Step 1: Open the console

Open the Devonfw console by executing the batch file `console.bat` from the Devonfw distribution. It is a pre-configured console which automatically uses the software and configuration provided by the Devonfw distribution.

[[getting-started-creating-new-devonfw-application\_step-2: change the directory]] ===== Step 2: Change the directory

You can create the project anywhere you want, but it is a good practice to create the projects in your workspace directory. Therefore, run the following command in the console to change to the directory to `workspaces\main`.

```
cd workspaces\main
```

This is the default location on Devonfw to create new applications. It is also possible to have your own workspace.

[[getting-started-creating-new-devonfw-application\_step-3: create the new application]] ===== Step 3: Create the new application

To create a new application, you need to execute one of the following commands:

- WAR packaging (arguments before archetype:generate identify the OASP4J archetype):

```
mvn -DarchetypeVersion=<OASP4J-VERSION> -DarchetypeGroupId=io.oasp.java.templates -  
DarchetypeArtifactId=oasp4j-template-server archetype:generate -DgroupId=<APPLICATION-GROUP-ID> -  
DartifactId=<APPLICATION-ARTIFACT-ID> -Dversion=<APPLICATION-VERSION> -Dpackage=<APPLICATION-PACKAGE-  
NAME>
```

For example

```
mvn -DarchetypeVersion=2.3.0 -DarchetypeGroupId=io.oasp.java.templates -DarchetypeArtifactId=oasp4j-template-server archetype:generate -DgroupId=io.oasp.application -DartifactId=sampleapp -Dversion=0.1-SNAPSHOT -Dpackage=io.oasp.application.sampleapp
```

This will create a new directory inside `workspaces\main` with the name of your application with the created application inside.

## 2.8.1 From Eclipse

### Note

If you use a proxy to connect to the Internet, then the above steps will not work as Eclipse has a known bug where the archetype discovery does not work behind a proxy. In this case, please use the command line version documented above.

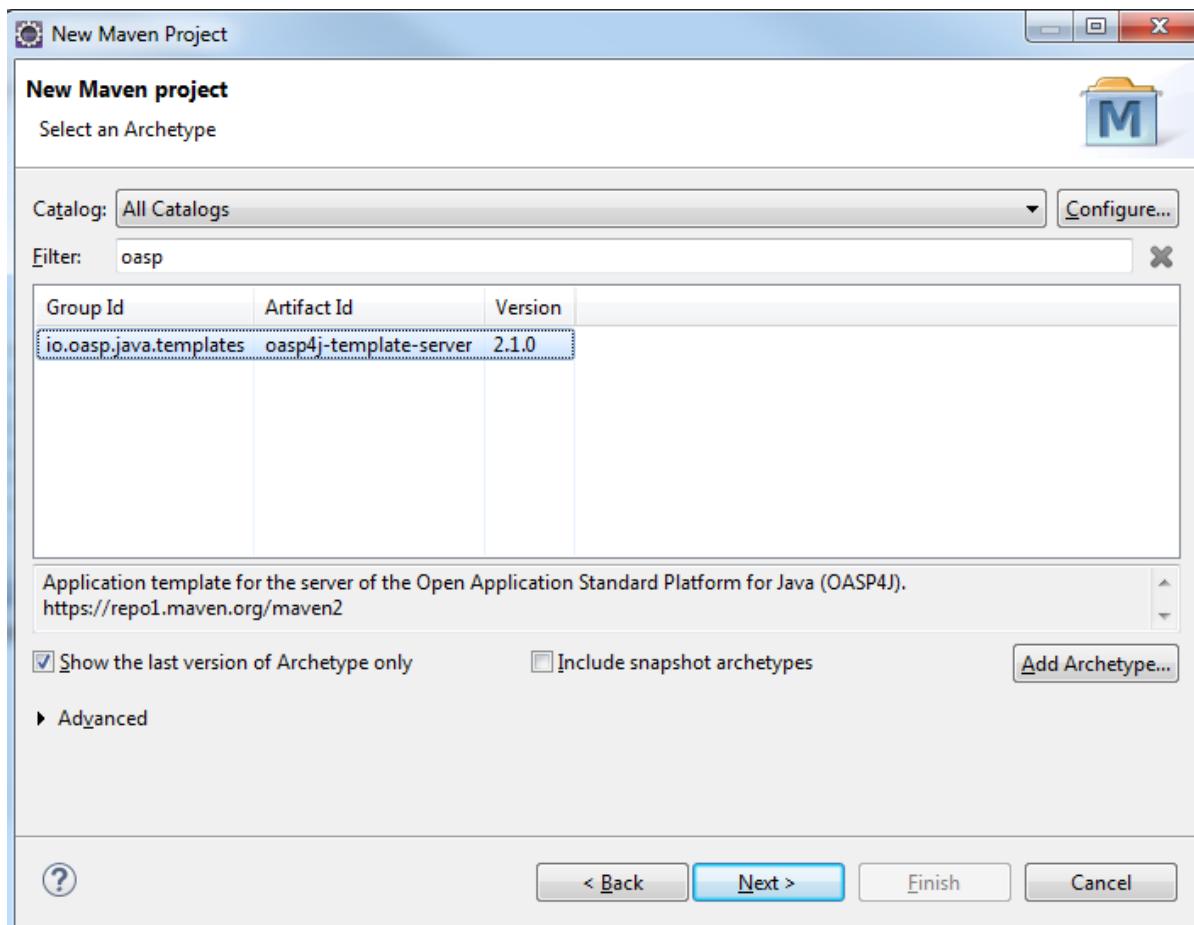
To create a new application using Eclipse, you should have installed [Devonfw distribution](#). Then, follow below steps to create a new application:

[[getting-started-creating-new-devonfw-application\_step-1 - create a new maven project]] ===== Step 1  
- Create a new Maven Project

Open Eclipse from a Devonfw distribution, by executing the batch file `eclipse-main.bat`, then go to *File > New > Maven Project*. If you don't see the option, click *File > New > Other* and use the filter to search the option *Maven Project*

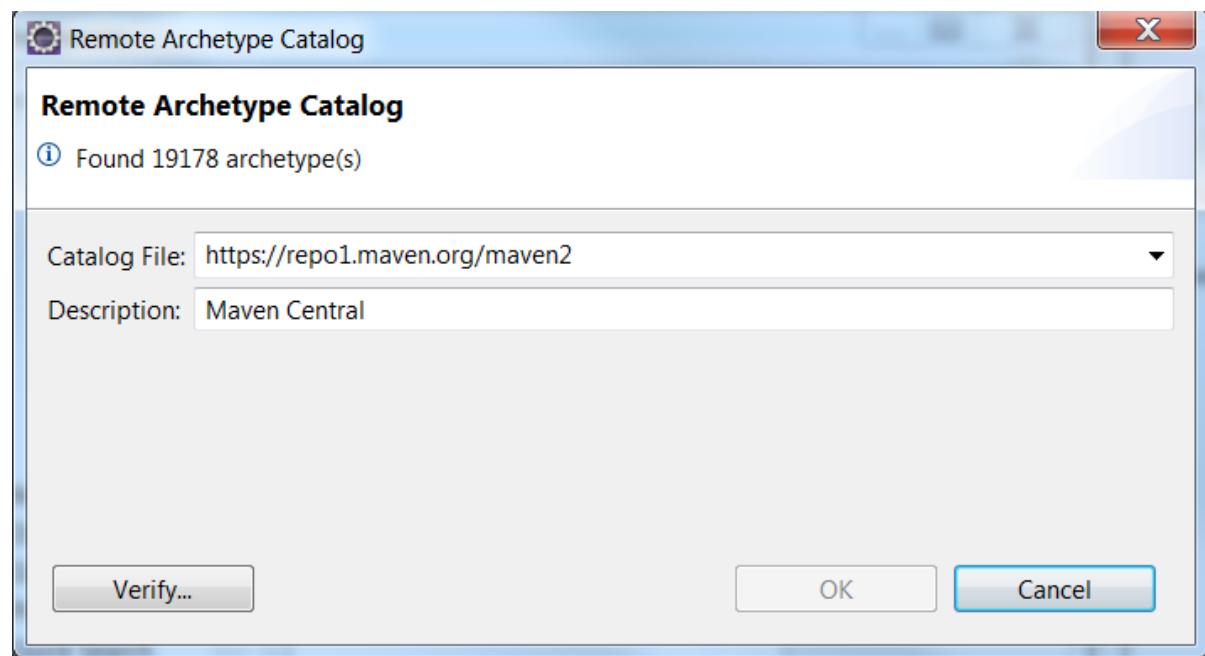
[[getting-started-creating-new-devonfw-application\_step-2 - choose the archetype]] ===== Step 2 - Choose the archetype

In the New Maven Project wizard, you need to choose the *oas4j-template-server archetype*, as shown in below image.



### Note

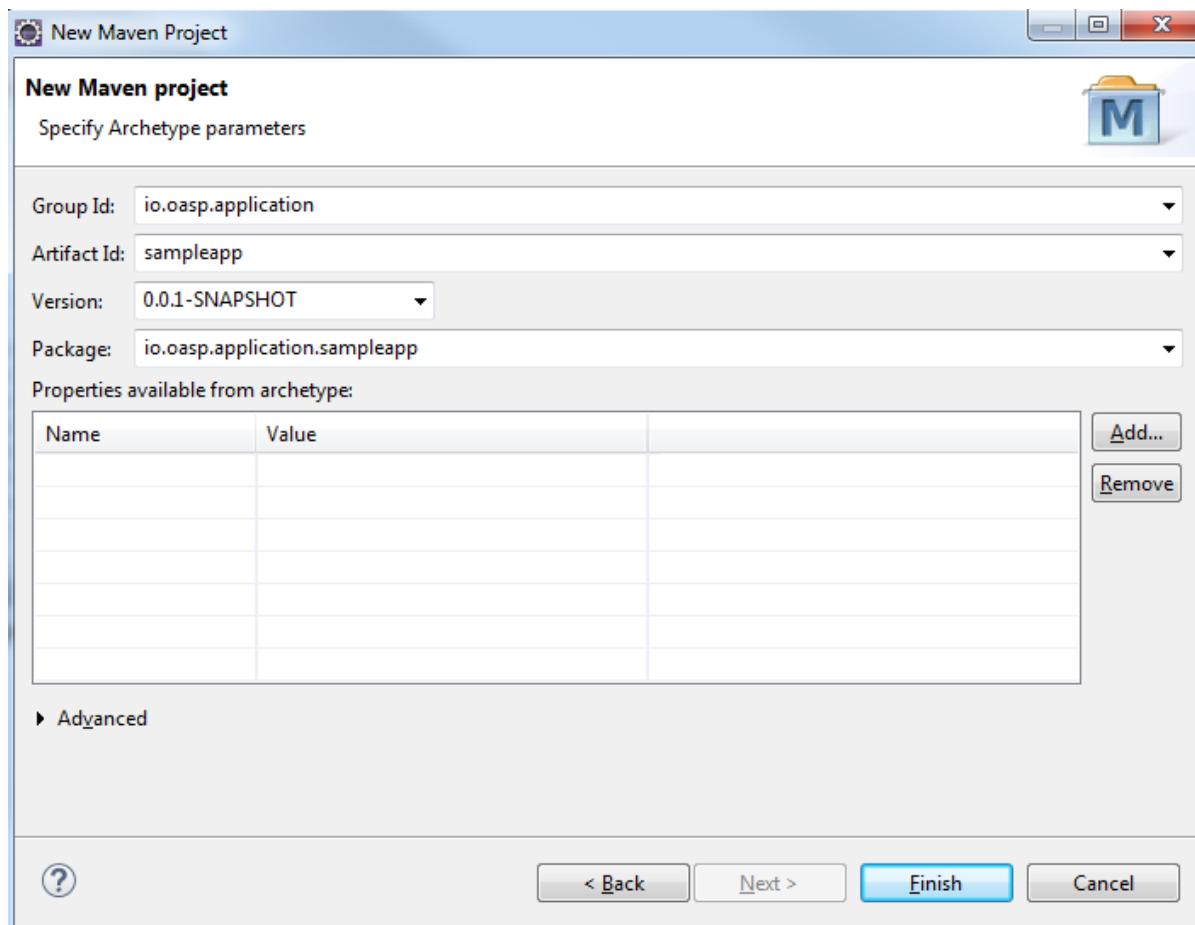
If you are not able to access the archetype even after checking the *Include snapshot archetypes* option, then try adding the archetype repository manually. You can do it with the *Configure* button located next to the *Catalogs* dropdown and then clicking the *Add Remote Catalog* button. Finally, you need to add the repository URL <https://repo1.maven.org/maven2> and as *Description* you can use *Maven Central*.



Use the *Verify* button to check the connection. Subsequently, you will see a message with the amount of found archetypes.

[[getting-started-creating-new-devonfw-application\_step-3 - configure the application properties]] =====  
Step 3 - Configure the application properties

Fill the *Group Id*, *Artifact Id*, *Version* and *Package* for your project.



- Click on the Finish button and the project will be ready for execution.

[[getting-started-creating-new-devonfw-application\_what-is generated]] == What is generated

To read more about the OASP4J application structure, click [here](#).

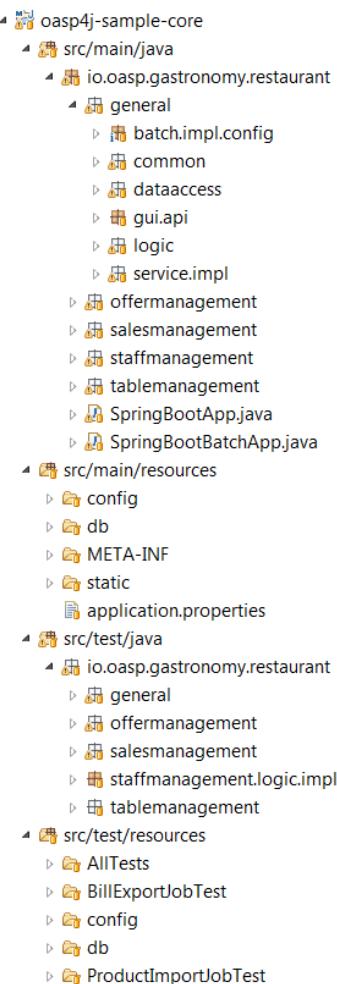
## 2.9 OASP4J Application Structure

The OASP4J archetype generates a Maven Modules Project. It has 3 Maven modules and a *pom.xml* file:

- **xxx-core:** This Maven Module Project contains the different Java classes and the application configuration (beans definition, security configuration, access to database configuration, etc).
- **xxx-server:** This module is responsible for packaging the application as a WAR file. It has a dependency on the core module.
- **pom.xml** is a configuration file that contains:
  - Properties definition: Spring-boot version, Java version, server port, database port.
  - Modules definition: Imports for modules (core and server).
  - Dependency Management: Spring-boot, OASP4J-BOM and JUnit dependencies. Also we can include Devonfw-BOM and Devonfw modules dependency.
  - Maven plugin compiler to compile the application to desired Java version.

### 2.9.1 Core Module

The Core module of the project contains the base classes and the base configuration of the application. It's divided in three main packages: *java*, *resources* and *test*.



Following section explains each of the package in detail:

## Java

The `java` package contains all Java files that implement the application and its configurations.

[[getting-started-oasp-app-structure\_springbootapp-and\_springbootbatchapp]] ===== `SpringBootApp` and `SpringBootBatchApp`

**SpringBootApp** is the class that contains the main method to run the Spring Boot application. Spring Boot is responsible to scan the project and find the beans and configuration of the application. Therefore, this is the most important class in the application as it's the basis for running the application. Read more about Spring Boot [here](#).

**SpringBootBatchApp** class is responsible to run the batch process of the application. Find more about the batch process in OASP4J project [here](#).

## General

The **general** package contains all Java files of OASP4J application. It is further divided in the following sub-packages:

[[getting-started-oasp-app-structure\_batch-impl config]] ===== Batch impl config

This first package stores a class `BeansBatchConfig.java` that contains the configuration of batch beans.

## Common

Contains the reusable Java components of OASP4J application.

## Dataaccess

This package contains the Java files that compose the Data Access layer of OASP4J application. Find more about this layer [here](#).

## Gui.api

Contains the Java class that has the login page controller of OASP4J application.

## Logic

Contains the logic layer of OASP4J application. The logic layer is the heart of the application and contains the main business logic. Find more about it [here](#).

## Service.impl.rest

Contains the service layer of OASP4J application. The service layer is responsible for exposing the functionality of the logical layer to external consumers over a network via technical protocols. Find more about it [here](#).

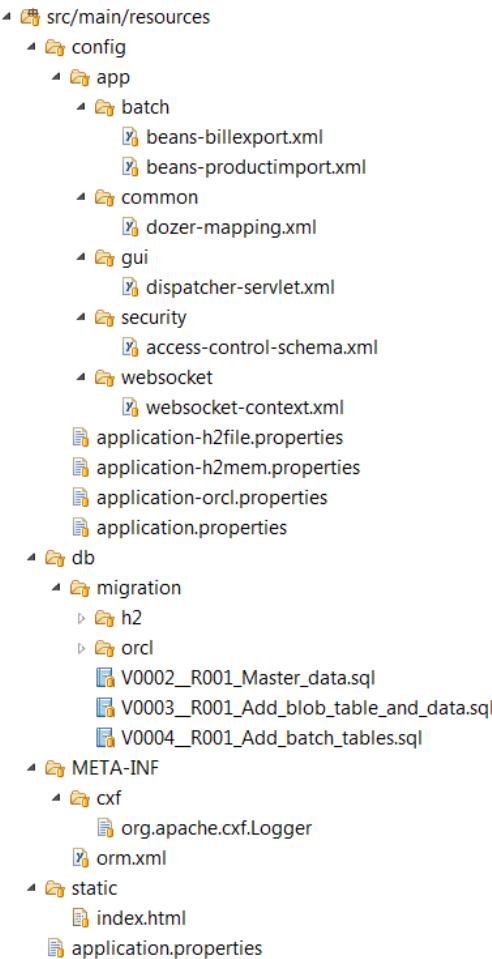
[[getting-started-oasp-app-structure\_management-packages ]] ===== Management packages

This packages contains the use cases in which the logic of the application is divided. Each management package defines the Entities, DAOs, ETOs, CTOs, REST services, web services and the CRUD operators related to its use case. Find more about how to create a CRUD operators in OASP4J project [here](#).

## Resources

The resources package contains the data related to some specific functionality of the application. This data can be both configuration files (database, security) and other resources that the application may need such as template files or content files.

This package is divided in four folders: *config*, *db*, *META-INF* and *static*. This package also contains **application.properties** file. Read more about OASP4J Spring Boot configuration [here](#).



## config

This folder contains mostly all the xml configuration file of the application. This folder also contains some **application.properties** files that represents the context of the application while running it with the embedded Tomcat server. See more about how to run an OASP4J application [here](#).

The **config** folder has a folder called **app** this folder is divided in the following sub-folders:

- **batch**: contains the configuration of the batch process. In the Sample Application, for example, we have the bill exports and products to import.
- **common**: contains the Spring bean configuration of [Dozer](#). find more about OASP4j Bean-Mapping [here](#).
- **gui**: this folder contains the **dispatcher-servlet.xml**. The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- **security**: contains the **access-control-schema.xml** file that contains the definition of groups/roles and permissions of the application.
- **websocket**: contains the scan component package definition for web sockets.

## db

This folder holds the SQL files which contains the script templates to create the database schema and tables definition.

## META-INF

Contains the **orm.xml** file that allows us to declare named queries that can be called in the code of the application to do a specific SQL queries.

## static

Contains the **index.html** file of the application. This view contains ,by default, a simple logout button and the link to a list of services of the application.

## application.properties

Contains the specific properties values of the application. This file is taken into account by the application when running in an [external server](#) (not the embedded).

As you can see we have an application-<name>.properties, this kind of properties are called profile and we can activate a determinate profile in **application.properties** as required. See more about Spring profiles [here](#). Also, find how to create a new database profile in OASP4j getting-started-database-configuration#create-a-spring-profile[here], in order to understand better.

## Test

The package test contains all that is required to test the application. It is divided in the following sub packages:

- java: contains the Unit Tests of the application that will allow us to keep control of the right functionality of the application. Find more about the Unit Test and TDD methodology [here](#) and [here](#)
- resources: contains the configuration and data that is required to run the test cases.

## 2.9.2 Server Module

This module contains two important files:

- lockback.xml: This file is in the *resources* folder and responsible for configuring the log.
- pom.xml: This file has Maven configuration for packaging the application as a WAR. Also, this file has a profile to package the JavaScript client ZIP file into the WAR.

## 2.10 Understanding the OASP4J and Spring Boot Configuration

An application needs to be configured in order to allow internal setup such as CDI (Context and Dependency Injection), but also to allow externalized configuration of a deployed package (e.g. integration into runtime environment). Using [Spring Boot](#), you can rely on a comprehensive configuration approach following a "convention over configuration" pattern. The Spring Boot guide adds on to this by detailed instructions and best-practices to deal with configurations.

In general, the kinds of configuration can be distinguished as explained in the following sections:

- [Internal Application configuration](#) maintained by developers
- [Externalized Environment configuration](#) maintained by the operators
- [Externalized Business configuration](#) maintained by business administrators

[[getting-started-understanding-oasp4j-spring-boot-config\_internal-application configuration]] == Internal Application Configuration

The application configuration contains all the internal settings and the wiring of the application (bean wiring, database mappings, etc.) and is maintained by the application developers at development time. Usually, there is a main configuration registered with the main Spring Boot App, but differing configurations to support automated test of the application can be defined using profiles (not detailed in this guide).

*web.xml* is referred as a place for all the web related configurations, but now it is no more used to configure the web app. It is empty. Therefore, let's discuss how you can configure the Devon based on Spring Boot applications!

[[getting-started-understanding-oasp4j-spring-boot-config\_standard-beans configuration]] == Standard beans configuration

For basic bean configuration, you can rely on spring boot, mainly using the configuration classes and occasionally xml-configuration files. Following are some of the key principles to understand Spring Boot auto-configuration features:

- Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies and annotated components found in your source code.
- Auto-configuration is noninvasive, at any point you can start to define your own configuration to replace specific parts of the auto-configuration by redefining your identically named bean.

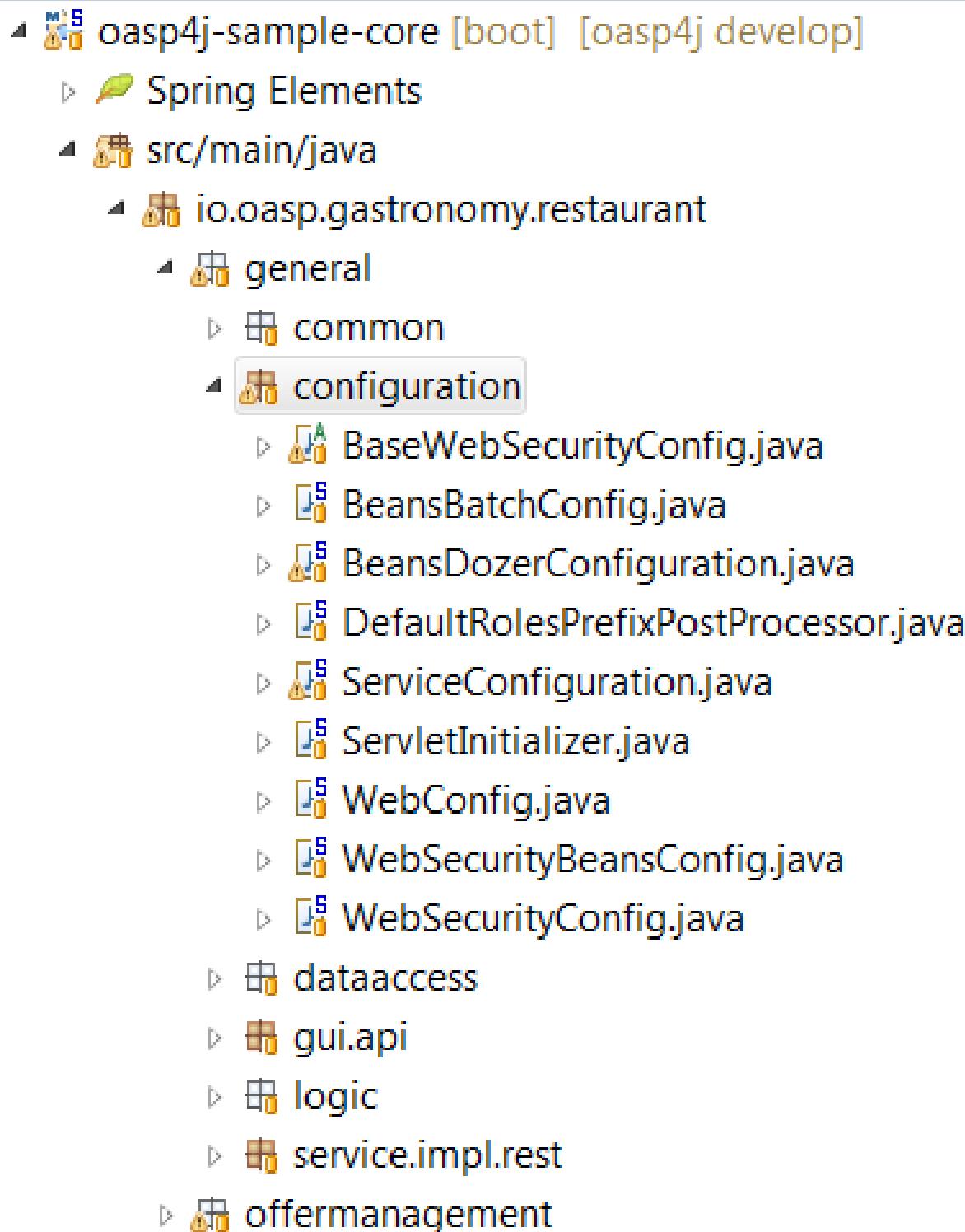
Beans are configured via annotations at the java-class (@Component, @Bean, @Named, etc.). These beans will be known when wiring the application at runtime. The required component scan is already auto-enabled within the main SpringBootApp.

For beans that needs separate configuration for any reason, additional Configuration Classes (using annotation @Configuration) can be used and will be automatically evaluated during application startup.

Lets see how you can customize your own Configuration Class

[[getting-started-understanding-oasp4j-spring-boot-config\_step-1: creating the configuration class]]  
==== Step 1: Creating the Configuration Class

In Devon, the Configuration Classes reside in the folder `src/main/general/configuration/` and it's recommended to include the new Configuration Classes here. This is just to keep a clear structure of projects. In fact, you can include the Configuration Classes anywhere in the project, it is the responsibility of the Spring Boot to scan the application.



Therefore, to create your Configuration Class, for example `src/main/general/configuration/MyConfigurationClass.java`

```
@Configuration
```

```
public class MyConfigurationClass{
    public int    propertiel;
    public String propertie2;
    public float  propertie3;

    private MyBean myBean;

    @Bean
    public MyBean myBean() {
        this.myBean= new MyBean(propertiel,propertie2,propertie3);
        return this.myBean;
    }
}
```

With the annotation `@Configuration` Spring Boot will manage your class as a configuration class.

[[getting-started-understanding-oasp4j-spring-boot-config\_step-2: including properties]] ===== Step 2:  
Including properties

In the last step, you have a Configuration Class that configures a simple bean (MyBean). Let's see how you can set values as per your configuration.

There are several ways of setting the parameter's value:

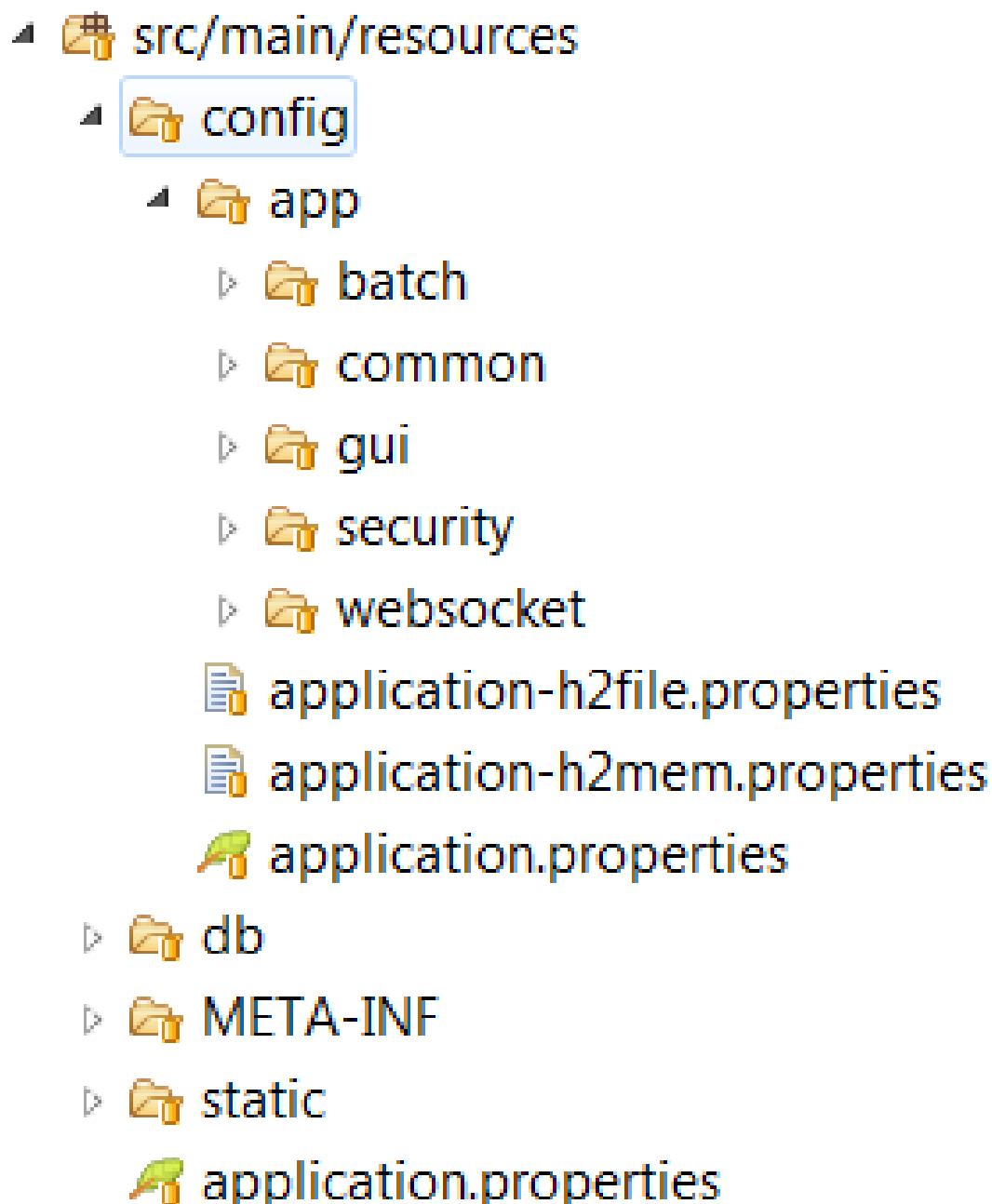
- Initialize the variable in the code.

```
(...)
public int    propertiel = 0;
public String propertie2 = "0";
public float  propertie3 = 0.0f;
(...)
```

Obviously, this is the simplest way to define the parameters, but isn't a good practice. Therefore, it is recommended to use some of the following ways to define the values in configuration.

- Using `@Value` annotation and `application.properties` file

First, you need to define the properties in the `application.properties` file. You can define the properties in `/main/resources/application.properties` or in `/main/resources/config/application.properties`. If you are running the server with the embedded Tomcat of the application, you can use both the files, but if you are deploying the application on an external Tomcat, you need to define your properties in the first one.



Learn more about how to run the application [here](#).

```
mybean.property1=0
mybean.property2=0
mybean.property3=0.0f
```

Finally, you can access the defined properties in the code using the `@Value` annotation:

```
(...)
@Value("${mybean.property1}")
public int property1;

@Value("${mybean.property2}")
public String property2;

@Value("${mybean.property3}")
public float property3;
```

( . . . )

- Using `@ConfigurationProperties` annotation and `application.properties` file

```
@Configuration
@ConfigurationProperties(prefix = "mybean")
public class MyConfigurationClass{
    public int property1;
    public String property2;
    public float property3;

    //WE NEED TO IMPLEMENT THE GETTERS AND SETTERS OF THE VARIABLES
}
```

Now, Spring Boot maps the variables to the value of the properties under the prefix "mybean". Therefore, you just need to include these in the `application.properties` file as you did in the `@Value` example.

[[getting-started-understanding-oasp4j-spring-boot-config\_xml-based-beans configuration]] === XML-based beans configuration It is still possible and allowed to provide (bean-) the configurations using xml, though not recommended. These configuration files are no more bundled via a main xml config file but loaded individually from their respective owners, e.g. for unit-tests:

```
@SpringApplicationConfiguration(classes = { SpringBootApp.class }, locations = { "classpath:/config/app/
batch/beans-productimport.xml" })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {
    ...
}
```

Configuration XML-files reside in an adequately named sub-folder of:

`src/main/resources/app`

## 2.10.1 Batch configuration

In the directory `src/main/resources/config/app/batch`, you can place the configuration file for the batch jobs. Each file within this directory represents one batch job.

## 2.10.2 WebSocket configuration

A websocket endpoint is configured within the business package as a Spring configuration class. The annotation `@EnableWebSocketMessageBroker` makes Spring Boot registering this endpoint.

```
package io.oasp.gastronomy.restaurant.salesmanagement.websocket.config;
...
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {
    ...
}
```

## 2.10.3 Externalized Configuration

Externalized configuration is provided separately in a deployment package and can be maintained undisturbed by redeployments.

### Environment Configuration

The environment configuration contains the configuration parameters (typically port numbers, host names, passwords, logins, timeouts, certificates, etc.) specifically for the different environments. These are under the control of the operators responsible for the application.

The environment configuration is maintained in the `application.properties` files, defining various properties. These properties are explained in the corresponding configuration sections of the guides for each topic:

- [persistence configuration](#)
- [service configuration](#)
- [logging guide](#)

There are two properties files exist within the example server:

- `src/main/resources/application.properties` provides a default configuration - bundled and deployed with the application package. It further acts as a template to derive a tailored minimal environment-specific configuration.
- `src/main/resources/config/application.properties` provides the additional properties only required at development time (for all local deployment scenarios). This property file is excluded from all packaging.

The location of the tailored `application.properties` file after deployment depends on the deployment strategy:

- standalone runnable Spring Boot App using embedded tomcat: place a tailored copy of `application.properties` into `installpath/config/`
- dedicated tomcat (one tomcat per app): place a tailored copy of `application.properties` into `tomcat/lib/config`
- tomcat serving a number of apps (requires expanding the wars): place a tailored copy of `application.properties` into the `tomcat/webapps/<app>/WEB-INF/classes/config`

In this `application.properties`, only define the minimum properties that are environment specific and inherit everything else from the bundled `src/main/resources/application.properties`. In any case, make sure that the class loader will find the file.

Also, assure that the properties are thoroughly documented by providing a comment to each property. This inline documentation is most valuable for your operations department.

## Business Configuration

The business configuration contains all business configuration values of the application, which can be edited by administrators through the GUI. The business configuration values are stored in the database in key/value pairs.

The database table `business_configuration` has the following columns:

- ID
- Property name
- Property type (Boolean, Integer, String)
- Property value
- Description

According to the entries in the above table, the administrative GUI shows a generic form to change business configuration. The hierarchy of the properties determines the place in the GUI, so the GUI bundles the properties from the same hierarchy level and name. Boolean values are shown as checkboxes, integer and string values as text fields. The properties are read and saved in a typed form, an error is raised if you try to save a string in an integer property, for example.

Following base layout is recommended for the hierarchical business configuration:

```
component.[subcomponent].[subcomponent].propertyname
```

## 2.11 Database Configuration

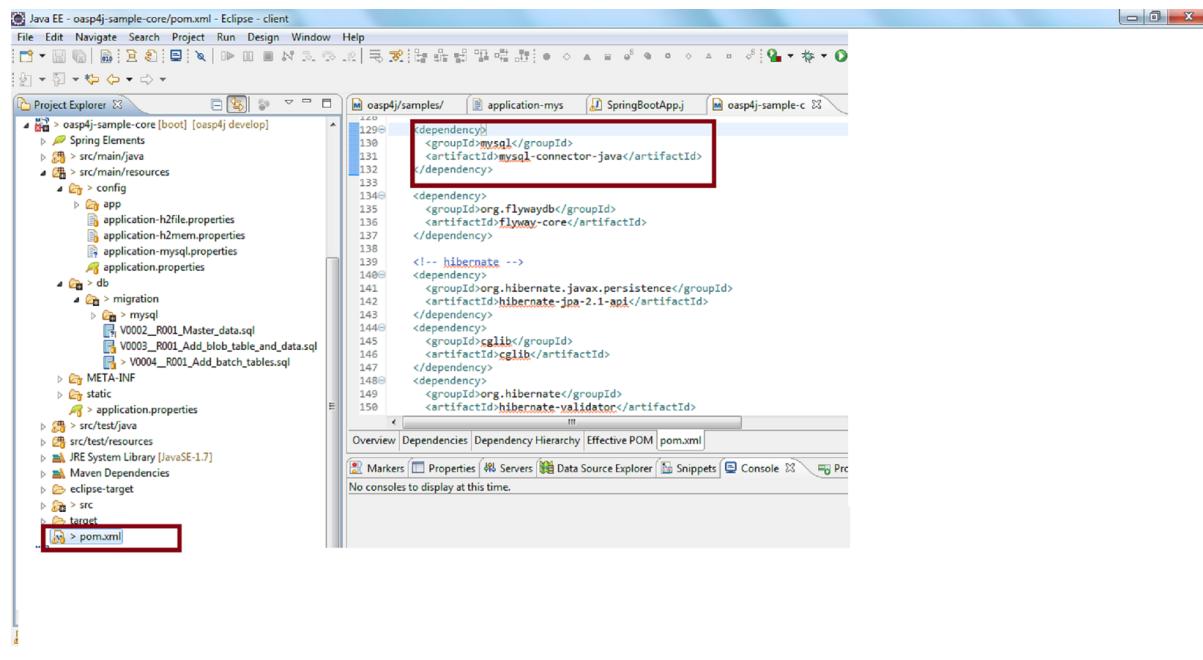
In this tutorial, you will see how to configure your application to connect with a real database of your choice. Devonfw gives you a simple configuration, so that you just need to add the dependency of the database libraries, create a profile, set the information about your database in the profile and activate it, update the script files, if any, as per DB syntax. All these steps need to be done in the generated (through oasp4j-template server archetype), `xxx-core` project. By default, it comes with configured [h2](#) database. If you want to configure another database such as MySQL, MS SQL or PostGre SQL, refer below sections.

### 2.11.1 Add dependencies

Add Dependency for database in `pom.xml` file. For example, to configure *mysql* database, add the following dependency:

MySQL:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```



**Note:** This driver should NOT be used in a production environment because of the license issues. See below for an alternative.

[[getting-started-database-configuration\_create-a spring profile]] == Create a Spring Profile

Spring Profile provides a way to segregate parts of your application configuration and make it only available in certain environments. In the above case, create a profile to your database configuration.

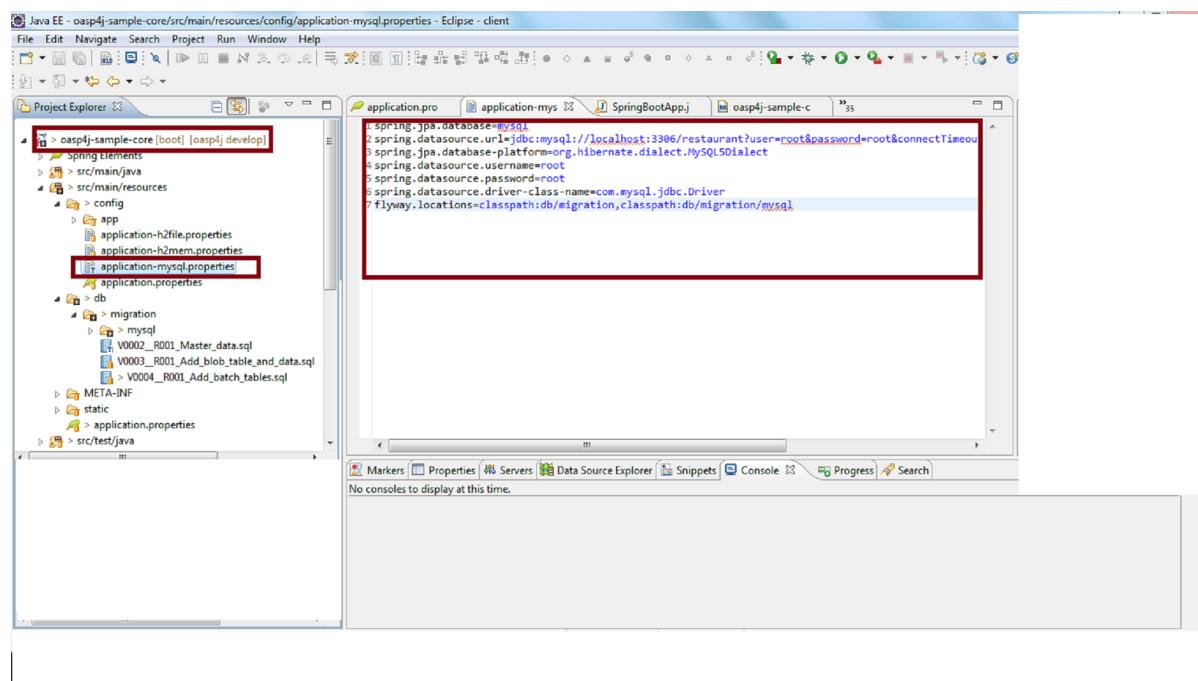
Create a file in `src/main/resources/config` named `application-{PROFILE_NAME}.properties`. The name of the profile isn't important. For this example, you can name it `application-mysql.properties`.

[[getting-started-database-configuration\_set-the\_database\_configuration]] == Set the database configuration

Add database configuration in *application-{PROFILE\_NAME}.properties* file. For example,

MySQL:

```
spring.jpa.database=mysql
spring.datasource.url=jdbc:mysql://localhost:3306/restaurant?
user=root&password=root&connectTimeout=10000&socketTimeout=10000&autoReconnect=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
flyway.locations=classpath:db/migration,classpath:db/migration/MySQL
```

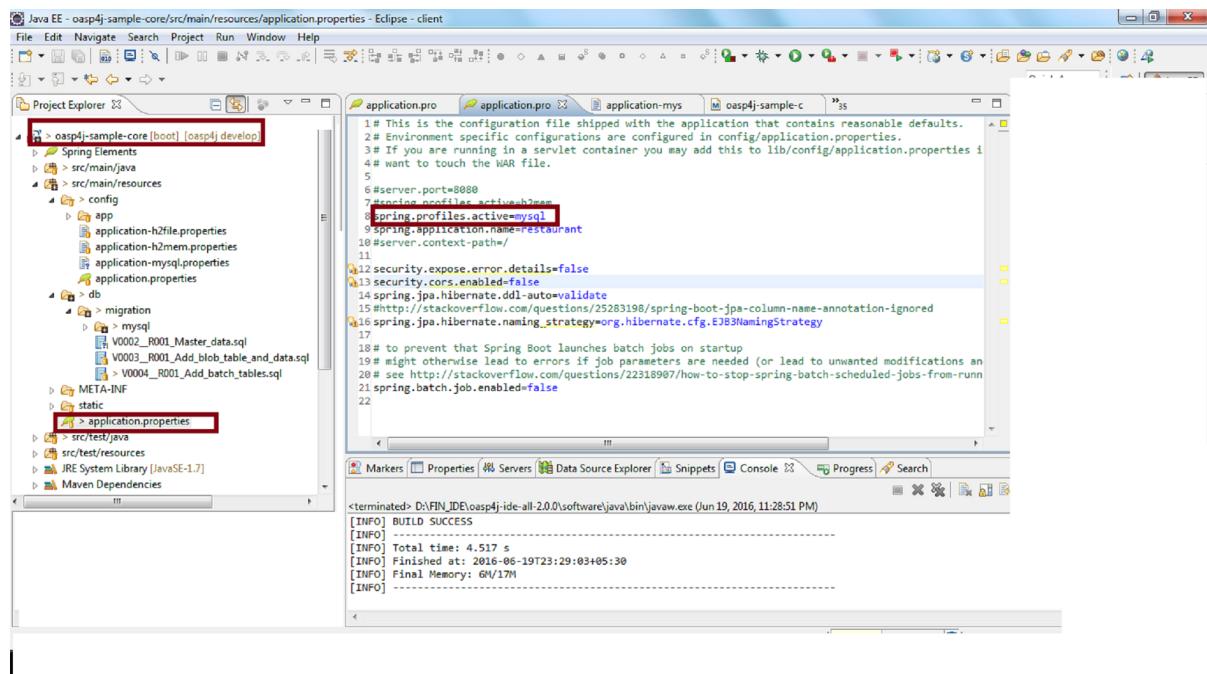


[[getting-started-database-configuration\_activate-the\_new\_profile]] == Activate the new profile

To activate the new profile, update `src/main/resources/application.properties` file. Set `spring.profiles.active=profile-name`. In the above example, the profile-name is *mysql*.

MySQL:

```
spring.profiles.active=mysql
```



[[getting-started-database-configuration\_further-details on database configurations]] == Further Details on Database Configurations

## MySQL

The use of the MySQL has already been illustrated in the above example. However, as mentioned, the GPL licensed (native) MySQL driver should **not** be used in a production environment. As an alternative, the free and liberally licensed "mariadb" (a MySQL clone) library could be used.

The dependency declaration consists of:

```
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
  <artifactId>mariadb-java-client</artifactId>
  <version>1.2.3</version>
</dependency>
```

And the library can be used such as MySQL but with a slight change in the configuration:

```
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
```

## PostgreSQL

The dependency declaration consists of:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.4-1206-jdbc41</version>
</dependency>
```

Ultimately, the following configuration must be used in order to use the postgresql driver and database:

```
spring.jpa.database=postgresql
spring.datasource.url=jdbc:postgresql://localhost:5432/<db name>
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.datasource.username=postgres
spring.datasource.password=<password set at time of installation>
```

```
spring.datasource.driver-class-name=org.postgresql.Driver
flyway.locations=classpath:db/migration,classpath:db/migration/postgres
```

[[getting-started-database-configuration\_microsoft-mssql server]] === Microsoft MSSQL Server

The Microsoft JDBC drivers are **not** available on Maven Central; [they need to be downloaded from the Microsoft site.](#)

Once downloaded, they should be installed in the local Maven repository (.m2 folder on the local machine). It can be done with the following command:

```
mvn install:install-file -DgroupId=com.microsoft.sqlserver -DartifactId=sqljdbc4 -Dversion=<version> -
Dpackaging=jar -DgeneratePom=true -Dfile=<driver JAR file>
```

Once installed, the library must be added to the project's *pom.xml* file. The dependency declaration should be something like

```
<dependency>
  <groupId>com.microsoft.sqlserver</groupId>
  <artifactId>sqljdbc4</artifactId>
  <version>4.0</version>
</dependency>
```

Ultimately, the following configuration must be used in order to use the MSSQL server driver and database:

```
spring.jpa.database=sqlserver
spring.datasource.url=jdbc:sqlserver://<servername>:<port>;databaseName=<databasename>
spring.jpa.database-platform=org.hibernate.dialect.SQLServerDialect
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver
```

(Replace *root* with your actual username / password)

For further information see: [MS SQL Server and MS JDBC Driver](#)

## DB2

The dependency with DB2 is explained below:

```
<dependency>
  <groupId>com.ibm.db2.jcc</groupId>
  <artifactId>db2jcc4</artifactId>
  <version>10.1</version>
</dependency>
<dependency>
  <groupId>com.ibm.db2</groupId>
  <artifactId>db2jcc_license_cisuz</artifactId>
  <version>9.7</version>
</dependency>
<dependency>
  <groupId>com.ibm.db2</groupId>
  <artifactId>db2java</artifactId>
  <version>9.7</version>
</dependency>
```

And the properties are explained below:

```
jdbc:db2://<server>:<port>/<dbalias>
spring.datasource.url=jdbc:db2://<server>:<port>:user=<user>;password=<password>;
spring.jpa.database-platform=org.hibernate.dialect.DB2Dialect
spring.datasource.traceLevel=com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL
spring.datasource.driver-class-name=com.ibm.db2.jcc.DB2Driver
```

---

You can see that the credential properties are included in the URL. If you want to learn more about URL format, you can see [SQLJ type 4 connectivity](#) and [SQLJ type 2 connectivity](#) URL syntax.

**Note**

The [IBM Drivers](#) are not freely distributed, so you can't find them in Maven. You need to contact IBM or just find the license in required IBM software product.

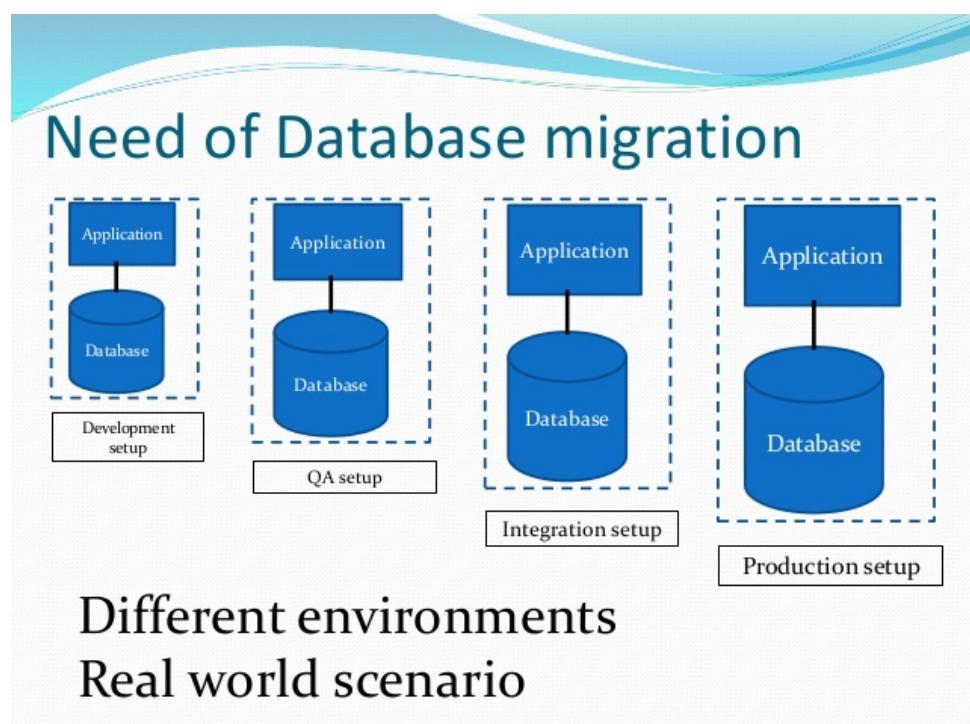
## 2.12 Set up and maintain database schemas with Flyway

[Flyway](#) is an open-source database migration tool. It strongly favors simplicity and convention over configuration.

[[getting-started-flyway-database-migration\_why-use flyway]] == Why use flyway

Consider, you have an application (a piece of software) and a database. Great! And this could be all you need.

But in most of the projects, this simple view of the world very quickly translates into this:



Then, you not only have to deal with one copy of the environment, but with several other. This presents a number of challenges to maintain the databases across various environments.

Many projects still rely on manually applied SQL scripts. And sometimes not even that (a quick SQL statement here or there to fix a problem). And soon many questions arise:

- What state is the database in on this machine?
- Has this script already been applied or not?
- Has the quick fix in production been applied in test afterwards?
- How do you set up a new database instance?

Most often the answer to above questions is: We don't know.

Database migration (where the flyway comes into the picture) is the great way to regain control over the above turmoil.

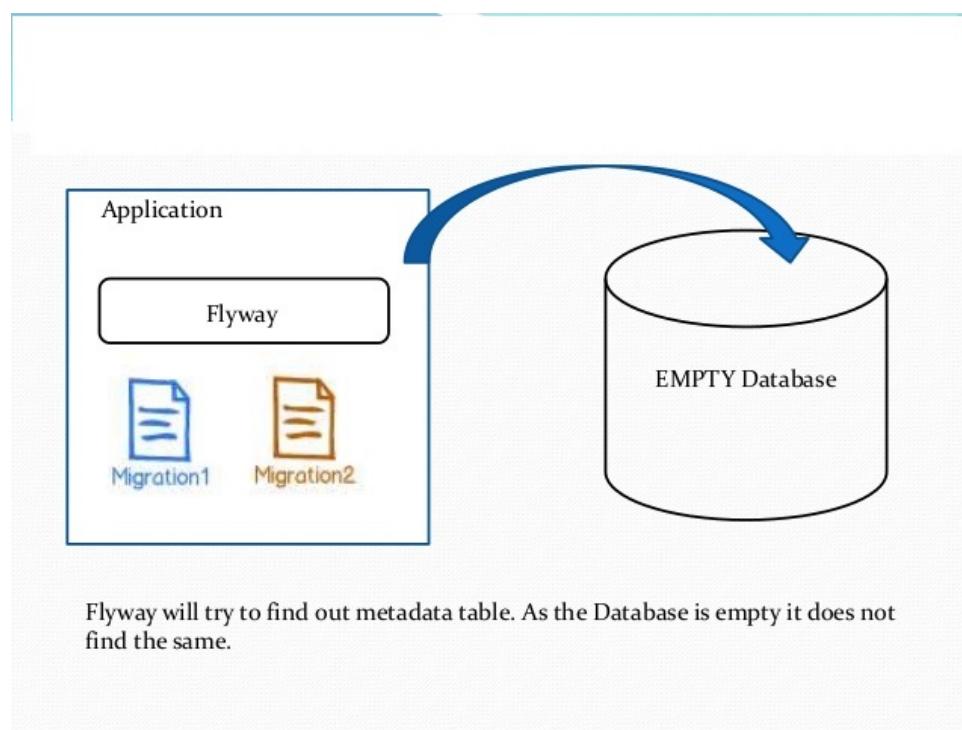
They allow you to:

- Recreate a database from scratch
- Make it clear at all times what state a database is in
- Migrate in a deterministic way from your current version of the database to a newer one.

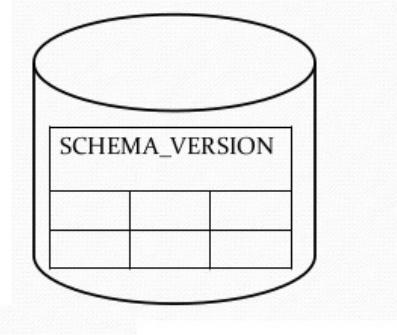
[[getting-started-flyway-database-migration\_how-it works for setting up the database and maintaining it]] == How it works for setting up the database and maintaining it

To know which state your database is in, Flyway relies on a special metadata table for all its internal bookkeeping.

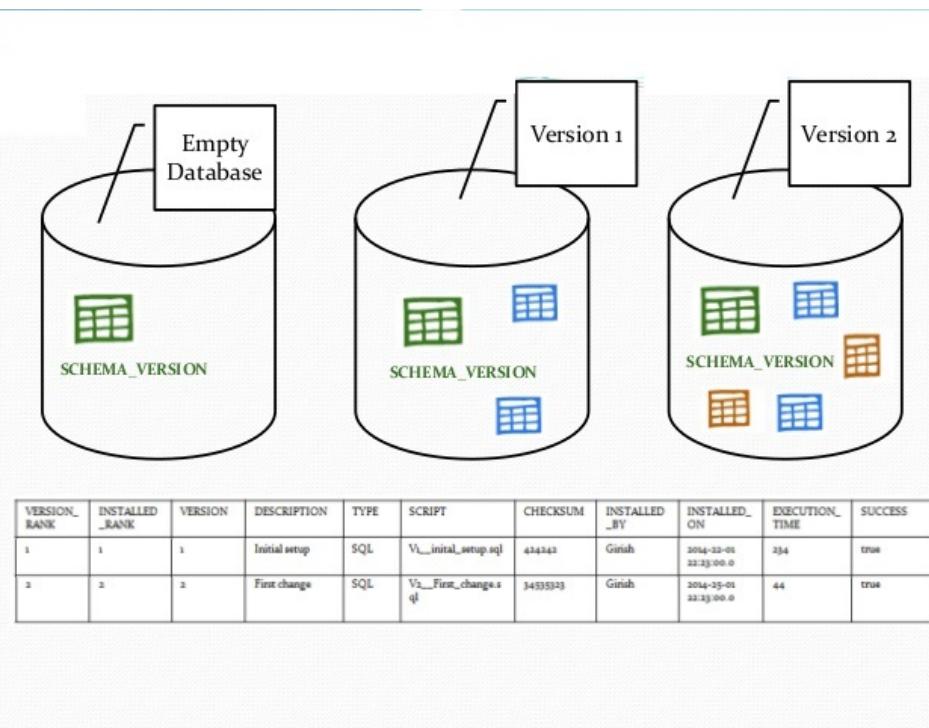
1. When you point Flyway to an empty database, it will try to locate its metadata table.



2. As the database is empty, Flyway won't find it and will create it instead. Now, you have a database with a single empty table called SCHEMA\_VERSION by default.



3. Immediately, Flyway will begin to scan the file system or the classpath of the application for migrations. They can be written in either SQL or Java.
4. The migrations are then sorted based on their version number and applied in order. As each migration gets applied, the metadata table is updated accordingly.



With the metadata and the initial state in place, you can now talk about migrating to newer versions.

Flyway will once again scan the filesystem or the classpath of the application for migrations. The migrations are checked against the metadata table. If their version number is lower or equal to the one of the versions marked as current, they are ignored.

And that's it! No matter when the need to evolve the database arises, whether structure (DDL) or reference data (DML), simply create a new migration with a version number higher than the current one. The next time Flyway starts, it will find it and upgrade the database accordingly.

A typical metadata table looks like below:

schema_version										
installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success	
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1996787037	axel	2016-02-04 22:23:00.0	546	true	
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-08 09:18:00.0	127	true	
3	2.1	Refactoring	JDBC	V2_1__Refactoring		axel	2016-02-10 17:45:05.4	251	true	

[[getting-started-flyway-database-migration\_how-to set up a database ]] == How to set up a database

Flyway can be used as a standalone or integrated tool via its API to make sure the database migration takes place on startup. To enable auto migration on startup (not recommended for production environment), set the following property in the *application.properties* file.

```
database.migration.auto = true
```

It is set to *false* by default via *application-default.properties* and shall be done explicitly in production environments. In a development environment, it is set to *true* in order to simplify development. This is also recommended for regular test environments.

If you want to use Flyway, set the following property in any case to prevent *Hibernate* from making changes in the database (pre-configured by default of OASp4J):

```
database.hibernate.hbm2ddl.auto=validate
```

If you want flyway to clear the database before applying the migrations (all data will be deleted), set the following property (default is *false*):

```
database.migration.clean = true
```

New database migrations are added to `src/main/resources/db/migrations`. They can be [SQL](#) based or [Java](#) based and follow below naming convention: `V<version>__<description>` (e.g.: `V0003__Add_new_table.sql`). For new SQL based migrations, stick to the following conventions:

- properties in camelCase
- tables in **UPPERCASE**
- ID properties with underscore (e.g. `table_id`)
- constraints all **UPPERCASE** with
- `PK_{table}` for primary key
- `FK_{sourcetable}2{target}` for foreign keys

- UC\_{table}\_{property} for unique constraints
- Inserts always with the same order of properties in blocks for each table
- Insert properties always starting with id, modificationCounter, [dtype, ] ...

For example, look at the sample script (migration) shown below:

```
-- *** Staffmember ***
CREATE TABLE STAFFMEMBER(
    id BIGINT NOT NULL,
    modificationCounter INTEGER NOT NULL,
    firstname VARCHAR(255),
    lastname VARCHAR(255),
    login VARCHAR(255),
    role INTEGER
);
```

It is also possible to use Flyway for test data. To do so, place your test data migrations in `src/main/resources/db/test-data/` and set property

```
database.migrationtestdata = true
```

Then, Flyway scans the additional location for migrations and applies all in the order specified by their version. If migrations `V_01...` and `V_02...` exist and a test data migration should be applied, in between, you can name it `V_01_1_...`.

## 2.13 Bean-Mapping using Dozer

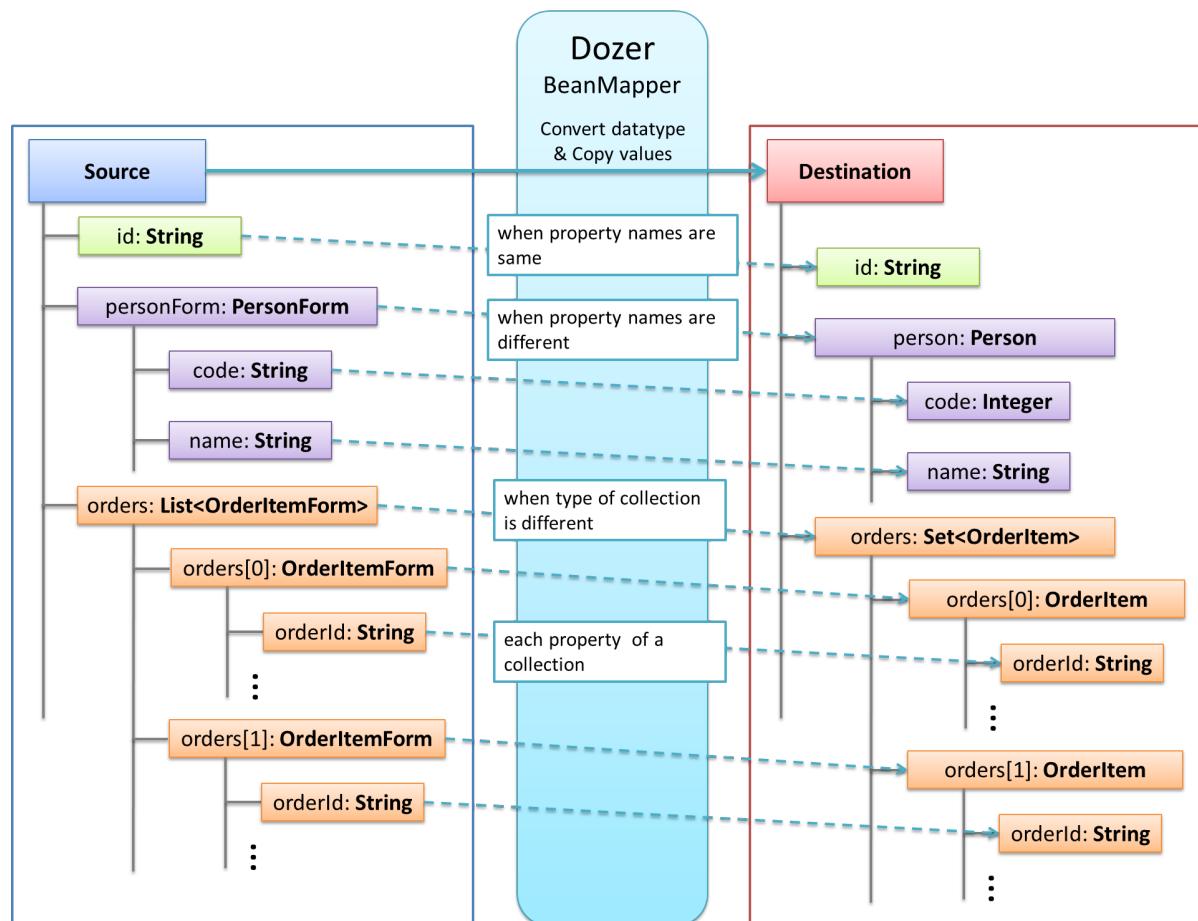
[[getting-started-bean-mapping-using-dozer\_why-use bean-mapping]] == Why use Bean-Mapping A mapping framework is useful in a layered architecture, where you can create layers of abstraction by encapsulating changes to particular data objects vs. propagating these objects to other layers (i.e. External service data objects, domain objects, data transfer objects, internal service data objects). A mapping framework is an ideal and can be used within Mapper type classes that are responsible for mapping data from one data object to another.

The challenge in distributed systems is passing the domain objects between different systems. Typically, you don't want internal domain objects to be exposed to the outside world and not allow external domain objects to bleed into your system.

Mapping between the data objects has been traditionally addressed by hand coding value object assemblers (or converters) that copy data between the objects. Most programmers will develop some sort of custom mapping framework and spend countless hours and thousands of lines of code mapping to and from their different data object.

A generic mapping framework solves these problems. Dozer (which is configured and used in Devonfw) is an open source mapping framework that is robust, generic, flexible, reusable, and configurable.

Typically, Dozer works as shown below:



For decoupling, you sometimes need to create separate objects (beans) for a different view. For example, for an external service, you will use a [transfer-object](#) instead of the [persistence entity](#), so internal changes to the entity do not implicitly change or break the service.

Therefore, you have the need to map similar objects which creates a copy. This is advantageous as the modifications to the copy has no side-effect on the original source object. However, to implement such mapping code by hand is very tedious and error-prone as shown below (if new properties are added to beans but not to mapping code):

```
public PersonTo mapPerson(PersonEntity source) {
    PersonTo target = new PersonTo();
    target.setFirstName(source.getFirstName());
    target.setLastName(source.getLastName());
    ...
    return target;
}
```

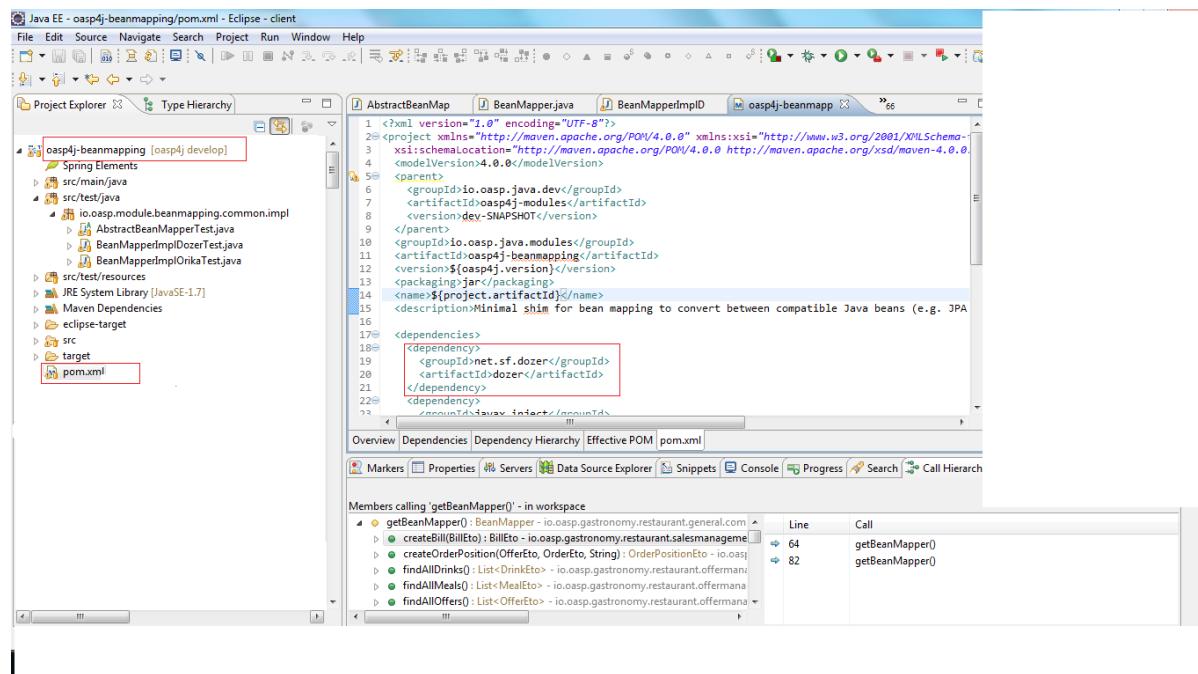
Therefore, *BeanMapper* is used for this purpose, which indirectly makes this task a lot easier.

### 2.13.1 Bean-Mapper Dependency

To get access to the *BeanMapper*, you can use this dependency in your *pom.xml* file:

```
<dependency>
  <groupId>io.oasp.java</groupId>
  <artifactId>oasp4j-beanmapping</artifactId>
</dependency>
```

So, (*oasp4j-beanmapping*) uses Dozer as dependency in its *pom.xml* file as shown below:



### 2.13.2 Bean-Mapper Usage

Then, you can get the *BeanMapper* via [dependency-injection](#) which is typically provided by an abstract base class (e.g. *AbstractUc*). Now, your problem can be solved easily:

```
PersonEntity person = ...;
...
return getBeanMapper().map(person, PersonTo.class);
```

---

So, in the above piece of code, `getBeanMapper()` method provides an mapper (dozer) instance , and when `map()` method is called, it maps `PersonEntity` (source object) to `PersonTo(DESTination object)`. Additionally, it supports the mapping of entire collections.

Dozer has been configured as a Spring bean in Devonfw, using dependency injection. This is done in `BeanDozerConfiguration.java` which is present in `resources/common/configuration` folder of `xxx-core` project, created using oasp4j template server archetype.

In this class, you can give path of mapping file (`dozer-mapping.xml`), which is generally placed at `config/app/common/dozer-mapping.xml`.

For more information on dozer, refer [here](#).

## 2.14 Write Unit Test Cases

### 2.14.1 Unit Test

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for the use. Intuitively, one can view a unit as the smallest testable part of an application. For more information, visit [wikipedia](#).

[[getting-started-writing-unittest-cases\_unit-test in eclipse]] === Unit Test in Eclipse

In order to understand how the Unit Tests works in Eclipse, lets discuss how to create and run a simple test.

[[getting-started-writing-unittest-cases\_step-1: create a new class]] === Step 1: Create a new class

Create a class with the name *MyClass.java* (you can [create a new application](#) as per need). In Eclipse, right click on the package in the application and then select *New > Class*. Name it *MyClass* and press *Finish*.

[[getting-started-writing-unittest-cases\_step-2: create a junit test]] === Step 2: Create a JUnit test

In Project Explorer, over the new *MyClass.java* class, then right click and go to *New > Other >* and select *JUnit Test Case*. Name it *MyClassTest* (name by default) and select source folder and package in the application to create the test. e.g. *src/test/java* (this is a good practice).

[[getting-started-writing-unittest-cases\_step-3: implement the test]] === Step 3: Implement the test First of all, check the dependencies of the module in *pom.xml* file.

```
<dependency>
  <groupId>io.oasp.java.modules</groupId>
  <artifactId>oasp4j-test</artifactId>
  <scope>test</scope>
</dependency>
```

In a OASP4J project, you have your own Component Test methods, so you need your new *JUnit Test* class to extend *AbstractComponentTest* class of the OASP4J module test.

```
@SpringBootTest(classes = { SpringBootApp.class })
public class MyClassTest extends AbstractComponentTest {

    @Test
    public void test() {
        assertThat(false).isTrue();
    }
}
```

This is a very simple test that verifies if the boolean value *true* is true. And is the case to start testing OASP4J application. As you can imagine that the test is going to fail, but you will see the details in later part.

#### Note

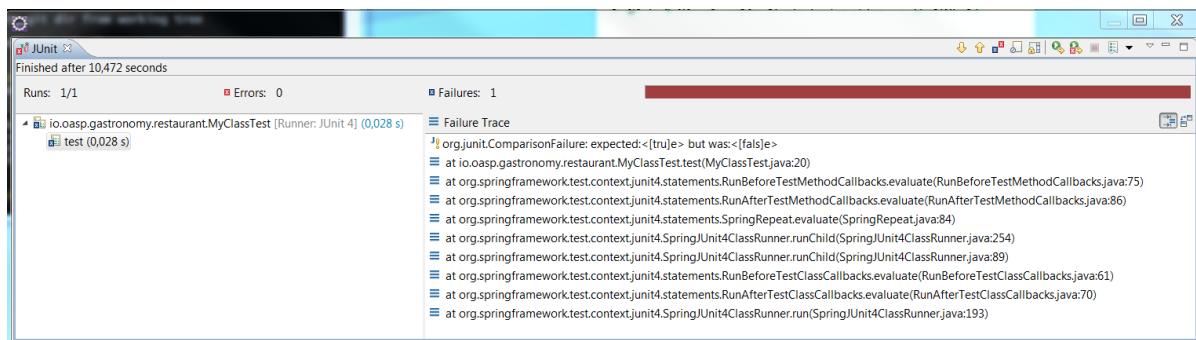
You are including the *@SpringBootTest* annotation in order to define an application context to your test. Without the context of the application, the test gets a fatal error, because you can't test a non-running application.

You can include a configuration location in the last annotation, if you need it, or use `@ContextConfiguration(locations = { "classpath:my-bean-context.xml" })`. For this tutorial, it's unnecessary because your test is the most simplest test you can perform.

[[getting-started-writing-unittest-cases\_step-4: run the test]] ===== Step 4: Run the test

Eclipse provides a very helpful view to test the applications. If you can't see, press the menu: *Windows* > *Show View* > *Other* and select *JUnit*.

Now, over the test, press right click *Run As* > *JUnit test*



In the above image, Eclipse shows a red rectangle because one of the tests has been failed (in this case, a single test). The failure of the test is marked with a blue cross but you can observe three different marks:

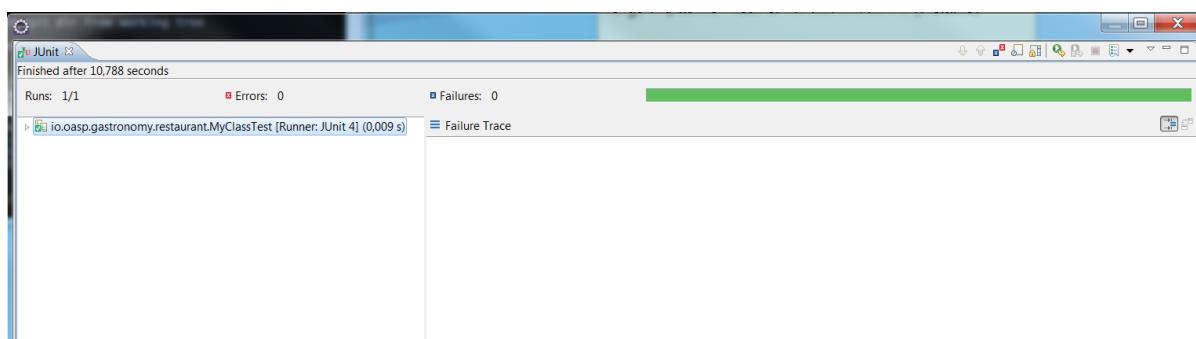
- Red cross: the test has some fatal error as, e.g context error, null pointer exceptions, etc.
- Blue cross: the test fails in some test method like `assertThat()` (like your case)
- Green check: the test is OK

In above case, you have a simple failure because your test has a `assertThat(false).isTrue()` meaning check if `true == false`. Now, let's discuss how to fix the failure and run the test again.

```
@SpringBootTest(classes = { SpringBootApp.class })
public class MyClassTest extends AbstractComponentTest {

    @Test
    public void test() {
        assertThat(true).isTrue();
    }
}
```

Now, you need to run the test again. And you will get the next result as shown in below image.



Evidently, the test ends successfully and Eclipse shows a green rectangle and the test with a green check.

With the discussed knowledge base, you can start testing all the applications.

[[getting-started-writing-unittest-cases\_tdd-test-driven development]] == TDD Test-driven development

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

The process of TDD is described as follows:

- Create a test
- Run all the tests
- Write the implementation code
- Run all the tests
- Refactor

[[getting-started-writing-unittest-cases\_tdd-in eclipse]] === TDD in Eclipse

Now, you are acquainted with the skills of creating, writing and running the test. Therefore, you can start with a simple tutorial in order to get the most clear idea about TDD.

The goal is create a simple calculator that has two methods: add(int,int) and sub(int,int).

[[getting-started-writing-unittest-cases\_step-1: create a test]] ===== Step 1: Create a test

The idea is very simple, you will create the tests for the methods of a class that needs to be implemented later. It will allow you to get the control of the result and verify that the code is working properly from the beginning.

You need to create a test called `CalculatorTest` in test package and a class `Calculator` in the java package.

In this test class, you will include a variable of a class `Calculator` and the test to the future `add()` and `sub()` methods of `Calculator` class.

Calculator.java

```
public class Calculator {  
    public Calculator() {}  
  
    public Object add(int a, int b) {  
        return null;  
    }  
  
    public Object sub(int a, int b) {  
        return null;  
    }  
}
```

Thus, you have the wire of your calculator. In this case, the implementation is very simple, but you can scale it to a more complex logic. Now, you need to include the test data required to run the class `CalculatorTest`.

### CalculatorTest.java

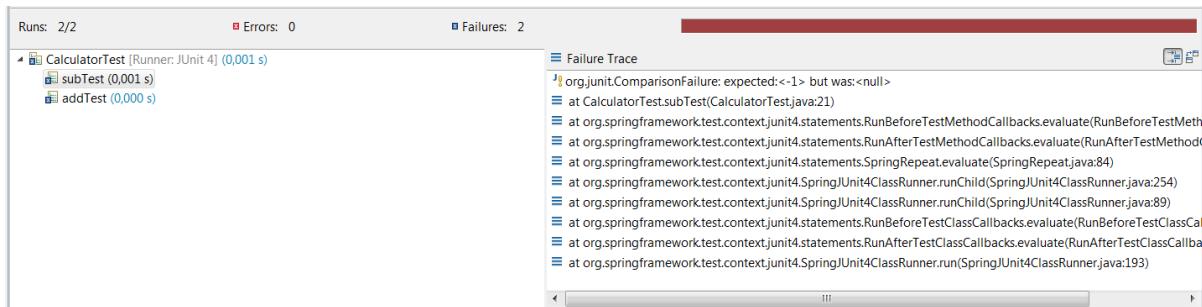
```
@SpringBootTest(classes = { SpringBootApp.class })
public class CalculatorTest extends AbstractComponentTest {
    private Calculator calculator = new Calculator();

    @Test
    public void addTest() {
        assertThat(this.calculator.add(1, 2)).isEqualTo(3);
    }

    @Test
    public void subTest() {
        assertThat(this.calculator.sub(1, 2)).isEqualTo(-1);
    }
}
```

[[getting-started-writing-unittest-cases\_step-2: run the test new test]] ===== Step 2: Run the test new test

Run the test and the result is as shown below:



Obviously, the test shows some failures as expected because the Calculator doesn't work yet.

The fact, this is more of a metaphoric step, as the implementation is in progress and it is obvious to get errors after running the test. As it is the cycle of the TDD, you need to write a test that will fail certainly so that the code to satisfy the test can be written. Surely, this will help to keep the code simple and clean.

#### Note

Methods named `add()` and `sub()`, returns `Object` as return value because if the methods return an `int`, you will get a "red cross error" pointing `NullPointerException` instead of "blue cross error" of `assertThat()`. It's just for this tutorial.

[[getting-started-writing-unittest-cases\_step-3: write the implementation code]] ===== Step 3: Write the implementation code

So far, you have seen a perfect test and an awful implementation of the Calculator. Let's start with the implementation.

Let's implement the method `add()` and see what happens.

```
public class Calculator {
    public Calculator() {}
```

```

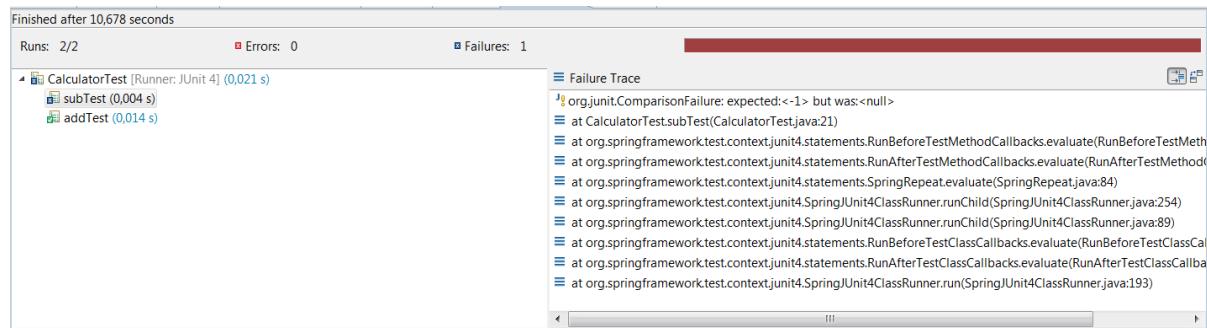
public int add(int a, int b) {
    return a + b;
}

public Object sub(int a, int b) {
    return null;
}
}

```

[[getting-started-writing-unittest-cases\_step-4: run the test -again-]] ===== Step 4: Run the test -again-

If you run the test, you will get the following result:



Now, you have a success result for the method `add()` and a failure result for the method `sub()`. Clearly, it's not necessary to get all the tests results as OK to run the tests, you can check the result of the test and work on to satisfy it. This the idea of TDD.

[[getting-started-writing-unittest-cases\_step-5: refactor]] ===== Step 5: Refactor

Now, let's implement the method `sub()`

```

public class Calculator {
    public Calculator() {}

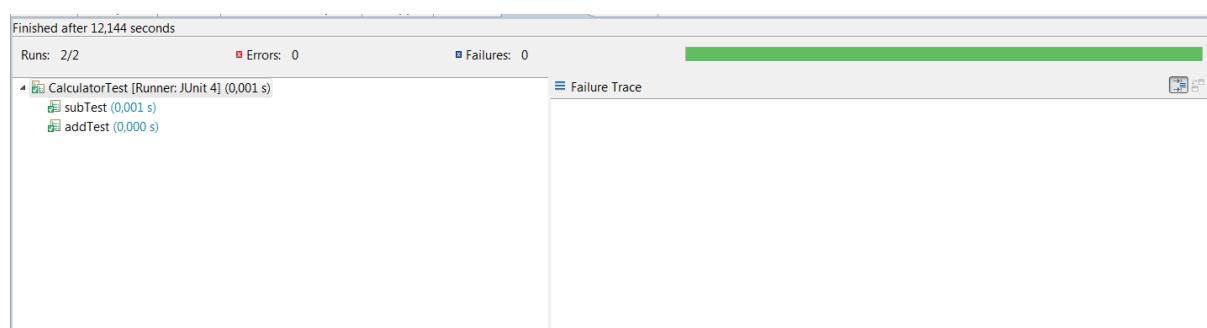
    public int add(int a, int b) {
        return a + b;
    }

    public int sub(int a, int b) {
        return a - b;
    }
}

```

[[getting-started-writing-unittest-cases\_step-6: run the test -return to step 2-]] ===== Step 6: Run the test -return to step 2-

If you run the application, you will get the following result:



---

Finally, here is your first application implemented with TDD methodology!

Therefore, in this tutorial, you have dealt with a very simple application, so you don't need another round of the TDD cycle, but in the real applications, you may need to repeat the cycle several times to get a successful result.

## 2.15 Logging and Auditing

### 2.15.1 Logging

We use [SLF4J](#) as API for logging. The recommended implementation is [Logback](#) for which we provide additional value such as configuration templates and an appender that prevents log-forging and reformatting of stack-traces for operational optimizations.

#### Usage

##### Maven Integration

In the *pom.xml* of your application add this dependency (that also adds transitive dependencies to SLF4J and logback):

```
<dependency>
  <groupId>io.oasp.java.modules</groupId>
  <artifactId>oasp4j-logging</artifactId>
  <version>2.3.0</version>
</dependency>
```

#### Configuration

The configuration file is *logback.xml* and is to put in the directory `src/main/resources` of your main application. For details consult the [logback configuration manual](#). OASP4J provides a production ready configuration [here](#). Simply copy this configuration into your application in order to benefit from the provided [operational](#) and aspects. We do not include the configuration into the *oasp4j-logging* module to give you the freedom of customizations (e.g. tune log levels for components and integrated products and libraries of your application).

The provided *logback.xml* is configured to use variables defined on the *config/application.properties* file. On our example, the log files path point to `../logs/` in order to log to tomcat log directory when starting tomcat on the bin folder. Change it according to your custom needs.

#### config/application.properties.

```
log.dir=../logs/
```

#### Logger Access

The general pattern for accessing loggers from your code is a static logger instance per class. We pre-configured the development environment so you can just type *LOG* and hit *[ctrl][space]* (and then *[arrow up]*) to insert the code pattern line into your class:

```
public class MyClass {
  private static final Logger LOG = LoggerFactory.getLogger(MyClass.class);
  ...
}
```

Please note that in this case we are not using injection pattern but use the convenient static alternative. This is already a common solution and also has performance benefits.

[[getting-started-logging-and-auditing\_how-to log]] ===== How to log We use a common understanding of the log-levels as illustrated by the following table. This helps for better maintenance and operation of the systems by combining both views.

Table 2.1. Loglevels

Loglevel	Description	Impact	Active Environments
FATAL	Only used for fatal errors that prevent the application to work at all (e.g. startup fails or shutdown/restart required)	Operator has to react immediately	all
ERROR	An abnormal error indicating that the processing failed due to technical problems.	Operator should check for known issue and otherwise inform development	all
WARNING	A situation where something worked not as expected. E.g. a business exception or user validation failure occurred.	No direct reaction required. Used for problem analysis.	all
INFO	Important information such as context, duration, success/failure of request or process	No direct reaction required. Used for analysis.	all
DEBUG	Development information that provides additional context for debugging problems.	No direct reaction required. Used for analysis.	development and testing
TRACE	Like DEBUG but exhaustive information and for code that is run very frequently. Will typically cause large log-files.	No direct reaction required. Used for problem analysis.	none (turned off by default)

Exceptions (with their stacktrace) should only be logged on *FATAL* or *ERROR* level. For business exceptions typically a *WARNING* including the message of the exception is sufficient.

## Operations

### Log Files

We always use the following log files:

- **Error Log:** Includes log entries to detect errors.
- **Info Log:** Used to analyze system status and to detect bottlenecks.

- **Debug Log:** Detailed information for error detection.

The log file name pattern is as follows:

```
<LOGTYPE>_log_<HOST>_<APPLICATION>_<TIMESTAMP>.log
```

Table 2.2. Segments of Logfilename

Element	Value	Description
<LOGTYPE>	info, error, debug	Type of log file
<HOST>	e.g. mywebserver01	Name of server, where logs are generated
<APPLICATION>	e.g. myapp	Name of application, which causes logs
<TIMESTAMP>	YYYY-MM-DD_HH00	date of log file

Example: *error\_log\_mywebserver01\_myapp\_2013-09-16\_0900.log*

Error log from *mywebserver01* at application *myapp* at 16th September 2013 9pm.

#### Output format

We use the following output format for all log entries to ensure that searching and filtering of log entries work consistent for all logfiles:

```
[D: <timestamp>] [P: <priority (Level)>] [C: <NDC>][T: <thread>][L: <logger name>]-[M: <message>]
```

- **D:** Date ( ISO8601: 2013-09-05 16:40:36,464)
- **P:** Priority (the log level)
- **C:** Correlation ID (ID to identify users across multiple systems, needed when application is distributed)
- **T:** Thread (Name of thread)
- **L:** Logger name (use class name)
- **M:** Message (log message)

Example:

```
[D: 2013-09-05 16:40:36,464] [P: DEBUG] [C: 12345] [T: main] [L: my.package.MyClass]-[M: My message...]
```

#### Security

In order to prevent [log forging](#) attacks we provide a special appender for logback in [oasp4j-logging](#). If you use it (see ) you are safe from such attacks.

[[getting-started-logging-and-auditing\_correlating-separate requests]] === Correlating separate requests In order to correlate separate HTTP requests to services belonging to the same user / session, we provide a servlet filter called "DiagnosticContextFilter". This filter first searches for a configurable HTTP header containing a correlation id. If none was found, it will generate a new correlation id. By default the HTTP header used is called "CorrelationId".

[[getting-started-logging-and-auditing\_auditing-with\_hibernate\_envers]] == Auditing with Hibernate Envers

For database auditing we use [hibernate envers](#). If you want to use auditing ensure you have the following dependency in your *pom.xml* file:

```
<dependency>
  <groupId>io.oasp.java.modules</groupId>
  <artifactId>oasp4j-jpa-envers</artifactId>
</dependency>
```

Make sure that entity manager (configured in *beans-jpa.xml*) also scans the package from the *oasp4j-jpa[-envers]* module in order to work properly.

```
...
<property name="packagesToScan">
  <list>
    <value>io.oasp.module.jpa.dataaccess.api</value>
  ...
</list>
```

Now let your DAO implementation extend from *AbstractRevisionedDao* instead of *AbstractDao* and your DAO interface extend from *[Application]RevisionedDao* instead of *[Application]Dao*.

The DAO now has a method *getRevisionHistory(entity)* available to get a list of revisions for a given entity and a method *load(id, revision)* to load a specific revision of an entity with the given ID.

To enable auditing for a entity simply place the *@Audited*annotation to your entity and all entity classes it extends from.

```
@Entity(name = "Drink")
@Audited
public class DrinkEntity extends ProductEntity implements Drink {
  ...
}
```

When auditing is enabled for an entity an additional database table is used to store all changes to the entity table and a corresponding revision number. This table is called *<ENTITY\_NAME>\_AUD* per default. Another table called *REVINFO* is used to store all revisions. Make sure that these tables are available. They can be generated by *Hibernate* with the following property (only for development environments).

```
database.hibernate.hbm2ddl.auto=create
```

Another possibility is to put them in your [database migration](#) scripts like so.

```
CREATE CACHED TABLE PUBLIC.REVINFO(
  id BIGINT NOT NULL generated by default as identity (start with 1),
  timestamp BIGINT NOT NULL,
  user VARCHAR(255)
);

CREATE CACHED TABLE PUBLIC.<TABLE_NAME>_AUD(
  <ALL_TABLE_ATTRIBUTES>,
  revtype TINYINT,
  rev BIGINT NOT NULL
);
```

## 2.16 Data-Access Layer

The data-access layer is responsible for all outgoing connections to access and process data. This is mainly about accessing data from a persistent data-store but also about invoking external services.

### 2.16.1 Persistence

For mapping java objects to a relational database we use the [Java Persistence API \(JPA\)](#). As JPA implementation we recommend to use [hibernate](#). For general documentation about JPA and hibernate follow the links above as we will not replicate the documentation. Here you will only find guidelines and examples about how we recommend to use it properly. The following examples show how to map the data of a database to an entity.

#### Entity

Entities are part of the persistence layer and contain the actual data. They are POJOs ([Plain Old Java Objects](#)) on which the relational data of a database is mapped and vice versa. The mapping is configured via JPA annotations (`javax.persistence`). Usually an entity class corresponds to a **table** of a database and a property to a **column** of that table. A persistent entity instance then represents a **row** of the database table.

[[getting-started-data-access-layer\_a-simple entity]] ===== A Simple Entity The following listing shows a simple example:

```
@Entity
@Table(name="TEXTMESSAGE")
public class MessageEntity extends AbstractPersistenceEntity {

    private String text;

    public String getText() {
        return this.text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

The `@Entity` annotation defines that instances of this class will be entities which can be stored in the database. The `@Table` annotation is optional and can be used to define the name of the corresponding table in the database. If it is not specified, the simple name of the entity class is used instead.

In order to specify how to map the attributes to columns we annotate the corresponding getter methods (technically also private field annotation is also possible but approaches can not be mixed). The `@Id` annotation specifies that a property should be used as [primary key](#).

```
@Entity
@Table(name="EMPLOYEE")
public class Employee
{
    private Long id;

    ...

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
```

```
    return this.id;
}
...
}
```

With the help of the `@Column` annotation it is possible to define the name of the column that an attribute is mapped to as well as other aspects such as `nullable` or `unique`. If no column name is specified, the name of the property is used as default.

```
@Column(name="DESCRIPTION", nullable=false, length=512)
public String getDescription() {
    return description;
}
```

Note that every entity class needs a constructor with public or protected visibility that does not have any arguments. Moreover, neither the class nor its getters and setters may be final.

Entities should be simple POJOs and not contain business logic.

#### [[getting-started-data-access-layer\_entities-and-datatypes]] ===== Entities and Datatypes

Standard datatypes like `Integer`, `BigDecimal`, `String`, etc. are mapped automatically by JPA. Custom [datatypes](#) are mapped as serialized [BLOB](#) by default what is typically undesired. In order to map atomic custom datatypes (implementations of `SimpleDatatype`) we implement an `AttributeConverter`. Here is a simple example:

```
@Converter(autoApply = true)
public class MoneyAttributeConverter implements AttributeConverter<Money, BigDecimal> {

    public BigDecimal convertToDatabaseColumn(Money attribute) {
        return attribute.getValue();
    }

    public Money convertToEntityAttribute(BigDecimal dbData) {
        return new Money(dbData);
    }
}
```

The annotation `@Converter` is detected by the JPA vendor if the annotated class is in the packages to scan (see `beans-jpa.xml`). Further, `autoApply = true` implies that the converter is automatically used for all properties of the handled datatype. Therefore all entities with properties of that datatype will automatically be mapped properly (in our example `Money` is mapped as `BigDecimal`).

In case you have a composite datatype that you need to map to multiple columns the JPA does not offer a real solution. As a workaround you can use a bean instead of a real datatype and declare it as [@Embeddable](#). If you are using hibernate you can implement `CompositeUserType`. Via the `@TypeDef` annotation it can be registered to hibernate. If you want to annotate the `CompositeUserType` implementation itself you also need another annotation (e.g. `MappedSuperclass` although not technically correct) so it is found by the scan.

#### Enumerations

By default JPA maps `Enums` via their ordinal. Therefore the database will only contain the ordinals (0, 1, 2, etc.) so inside the database you can not easily understand their meaning. Using `@Enumerated` with `EnumType.STRING` allows to map the `enum` values to their name (`Enum.name()`). Both approaches are fragile when it comes to code changes and refactorings (if you change the order of the `enum` values or rename them) after being in production with your application. If you want to avoid this and get a robust

mapping you can define a dedicated string in each *enum* value for database representation that you keep untouched. Then you treat the *enum* just like any other [custom datatype](#).

## BLOB

If binary or character large objects (BLOB/CLOB) should be used to store the value of an attribute, e.g. to store an icon, the `@Lob` annotation should be used as shown in the following listing:

```
@Lob
public byte[] getIcon() {
    return this.icon;
}
```

### Warning

Using a byte array will cause problems if BLOBs get large because the entire BLOB is loaded into the RAM of the server and has to be processed by the garbage collector. For larger BLOBs the type [Blob](#) and streaming should be used.

```
public Blob getAttachment() {
    return this.attachment;
}
```

[[getting-started-data-access-layer\_date-and time]] ===== Date and Time To store date and time related values, the `@Temporal`/annotation can be used as shown in the listing below:

```
@Temporal(TemporalType.TIMESTAMP)
public java.util.Date getStart() {
    return start;
}
```

Until Java8 the java data type `java.util.Date` (or [Jodatime](#)) has to be used. `TemporalType` defines the granularity. In this case, a precision of nanoseconds is used. If this granularity is not wanted, `TemporalType.DATE` can be used instead, which only has a granularity of milliseconds. Mixing these two granularities can cause problems when comparing one value to another. This is why we **only** use `TemporalType.TIMESTAMP`.

## Primary Keys

We only use simple Long values as primary keys (IDs). By default it is auto generated (`@GeneratedValue(strategy=GenerationType.AUTO)`). This is already provided by the class `io.oasp.module.jpa.persistence.api.AbstractPersistenceEntity` that you can extend. In case you have business oriented keys (often as `String`), you can define an additional property for it and declare it as unique (`@Column(unique=true)`).

[[getting-started-data-access-layer\_data-access object]] === Data Access Object *Data Access Objects* (DAOs) are part of the persistence layer. They are responsible for a specific [entity](#) and should be named `<Entity>Dao[Impl]`. The DAO offers the so called CRUD-functionalities (create, retrieve, update, delete) for the corresponding entity. Additionally a DAO may offer advanced operations such as query or locking methods.

## DAO Interface

For each DAO there is an interface named `<Entity>Dao` that defines the API. For CRUD support and common naming we derive it from the interface `io.oasp.module.jpa.persistence.api.Dao`:

```
public interface MyEntityDao extends Dao<MyEntity> {  
  
    List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria);  
}
```

As you can see, the interface *Dao* has a type parameter for the entity class. All CRUD operations are only inherited so you only have to declare the additional methods.

## DAO Implementation

Implementing a DAO is quite simple. We create a class named *<Entity>DaoImpl* that extends *io.oasp.module.jpa.persistence.base.AbstractDao* and implements your *<Entity>Dao* interface:

```
public class MyEntityDaoImpl extends AbstractDao<MyEntity> implements MyEntityDao {  
  
    public List<MyEntity> findByCriteria(MyEntitySearchCriteria criteria) {  
        TypedQuery<MyEntity> query = createQuery(criteria, getEntityManager());  
        return query.getResultList();  
    }  
    ...  
}
```

As you can see *AbstractDao* already implements the CRUD operations so you only have to implement the additional methods that you have declared in your *<Entity>Dao* interface. In the DAO implementation you can use the method *getEntityManager()* to access the *EntityManager* from the JPA. You will need the *EntityManager* to create and execute [queries](#).

## Queries

The [Java Persistence API \(JPA\)](#) defines its own query language, the java persistence query language (JPQL), which is similar to SQL but operates on entities and their attributes instead of tables and columns.

### Static Queries

The OASP4J advises to specify all queries in one mapping file called *orm.xml* (located in *src/main/resources/META-INF* directory).

You can add the new queries to this file:

```
<?xml version="1.0" encoding="UTF-8"?>  
<entity-mappings version="1.0" xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://  
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm  
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">  
...  
    <named-query name="get.staff.member.by.login">  
        <query><![CDATA[SELECT s FROM StaffMemberEntity s WHERE login = :login]]></query>  
    </named-query>  
...  
</entity-mappings>
```

To avoid redundant occurrences of the query name (*get.staff.member.by.login*) we define the constants for each named query:

```
package io.oasp.gastronomy.restaurant.general.common.api.constants;  
  
public class NamedQueries {  
...  
    public static final String GET_STAFF_MEMBER_BY_LOGIN = "get.staff.member.by.login";  
...  
}
```

Note that changing the name of the java constant (`GET_STAFF_MEMBER_BY_LOGIN`) can be easily done with refactoring (in Eclipse right click over the property and select *Refactor > Rename*). Further you can trace where the query is used by searching the references of the constant.

The following listing shows how to use this query in class `StaffMemberDaoImpl` (remember that we must adapt `StaffMemberDao`).

We will have a `StaffMemberDao` like the following:

```
public interface StaffMemberDao extends ApplicationDao<StaffMemberEntity>,
MasterDataDao<StaffMemberEntity> {
    StaffMemberEntity findByLogin(String login);
    ...
}
```

And the implementation of the interface would be:

```
public class StaffMemberDaoImpl extends ApplicationMasterDataDaoImpl<StaffMemberEntity> implements
StaffMemberDao {
    ...
    @Override
    public StaffMemberEntity findByLogin(String login) {
        TypedQuery<StaffMemberEntity> query =
            getEntityManager().createNamedQuery(NamedQueries.GET_STAFF_MEMBER_BY_LOGIN,
StaffMemberEntity.class);
        query.setParameter("login", login);
        return query.getSingleResult();
    }
    ...
}
```

The `EntityManager` contains a method called `createNamedQuery(String)`, which takes as parameter the name of the query and creates a new query object. The parameters of the query have to be set using the `setParameter(String, Object)` method.

### Note

Using the `createQuery(String)` method, which takes as parameter the query (a string that already contains the parameters), is not allowed as this makes the application vulnerable to SQL injection attacks.

When the method `getResultSet()` is invoked, the query is executed and the result is delivered as list. As an alternative, there is a method called `getSingleResult()`, which returns the entity if the query returned exactly one and throws an exception otherwise.

[[getting-started-data-access-layer\_using-queries to avoid bidirectional relationships ]] ===== Using Queries to Avoid Bidirectional Relationships With the usage of queries it is possible to avoid bidirectional relationships, which have some disadvantages (see [relationships](#)). So for example to get all `WorkingTime`'s for a specific `StaffMember` without having an attribute in the `StaffMember`'s class that stores these `WorkingTime`'s, the following query is needed:

```
<named-query name="working.time.search.by.staff.member">
```

```
<query><![CDATA[select work from WorkingTime where work.staffMember = :staffMember]]></query>
</named-query>
```

The method looks as follows (extract of class *WorkingTimeDaoImpl*):

```
public List<WorkingTime> getWorkingTimesForStaffMember(StaffMember staffMember) {
    Query query = getEntityManager().createNamedQuery(NamedQueries.WORKING_TIMES_SEARCH_BY_STAFFMEMBER);
    query.setParameter("staffMember", staffMember);
    return query.getResultList();
}
```

Do not forget to adapt the *WorkingTimeDao* interface and the *NamedQueries* class accordingly.

To get a more detailed description of how to create queries using JPQL, please have a look [here](#) or [here](#).

## Dynamic Queries

For dynamic queries we recommend to use [QueryDSL](#). It allows to implement queries in a powerful but readable and type-safe way (unlike Criteria API). If you already know JPQL you will quickly be able to read and write QueryDSL code. It feels like JPQL but implemented in Java instead of plain text.

Please be aware that code-generation can be painful especially with large teams. We therefore recommend to use QueryDSL without code-generation. Here is an example from our sample application:

```
public List<OrderEntity> findOrders(OrderSearchCriteriaTo criteria) {

    OrderEntity order = Alias.alias(OrderEntity.class);
    EntityPathBase<OrderEntity> alias = Alias.$(order);
    JPAQuery query = new JPAQuery(getEntityManager()).from(alias);
    Long tableId = criteria.getTableId();
    if (tableId != null) {
        query.where(Alias.$(order.getTableId()).eq(tableId));
    }
    OrderState state = criteria.getState();
    if (state != null) {
        query.where(Alias.$(order.getState()).eq(state));
    }
    applyCriteria(criteria, query);
    return query.list(alias);
}
```

## Using Wildcards

For flexible queries it is often required to allow wildcards (especially in [dynamic queries](#)). While users intuitively expect glob syntax the SQL and JPQL standards work different. Therefore a mapping is required (see [here](#)).

## Pagination

The OASP4J provides the method *findPaginated* in *AbstractGenericDao* that executes a given query (for now only QueryDSL is supported) with pagination parameters based on *SearchCriteriaTo*. So all you need to do is derive your individual search criteria objects from *SearchCriteriaTo*, prepare a QueryDSL-query with the needed custom search criterias, and call *findPaginated*. Here is an example from our sample application:

```
@Override
public PaginatedListTo<OrderEntity> findOrders(OrderSearchCriteriaTo criteria) {

    OrderEntity order = Alias.alias(OrderEntity.class);
    EntityPathBase<OrderEntity> alias = $(order);
    JPAQuery query = new JPAQuery(getEntityManager()).from(alias);
```

```
Long tableId = criteria.getTableId();
if (tableId != null) {
    query.where($($order.getTableId()).eq(tableId));
}
OrderState state = criteria.getState();
if (state != null) {
    query.where($($order.getState()).eq(state));
}

return findPaginated(criteria, query, alias);
}
```

Then the query allows pagination by setting `pagination.size` (`SearchCriteriaTo.getPagination().setSize(Integer)`) to the number of hits per page and `pagination.page` (`SearchCriteriaTo.getPagination().setPage(int)`) to the desired page. If you allow the client to specify `pagination.size`, it is recommended to limit this value on the server side (`SearchCriteriaTo.limitMaximumPageSize(int)`) to prevent performance problems or DOS-attacks. If you need to also return the total number of hits available, you can set `SearchCriteria.getPagination().setTotal(boolean)` to `true`.

### Pagination example

For the table entity we can make a search request by accessing the REST endpoint with pagination support like in the following examples:

```
POST oasp4j-sample-server/services/rest/tablemanagement/v1/table/search
{
    "pagination": {
        "size": 2,
        "total": true
    }
}

//Response
{
    "pagination": {
        "size": 2,
        "page": 1,
        "total": 11
    },
    "result": [
        {
            "id": 101,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 1,
            "state": "OCCUPIED"
        },
        {
            "id": 102,
            "modificationCounter": 1,
            "revision": null,
            "waiterId": null,
            "number": 2,
            "state": "FREE"
        }
    ]
}
```

### Note

as we are requesting with the `total` property set to `true` the server responds with the total count of rows for the query.

For retrieving a concrete page, we provide the `page` attribute with the desired value. Here we also left out the `total` property so the server doesn't incur on the effort to calculate it:

```
POST oasp4j-sample-server/services/rest/tablemanagement/v1/table/search
{
  "pagination": {
    "size": 2,
    "page": 2
  }
}

//Response

{
  "pagination": {
    "size": 2,
    "page": 2,
    "total": null
  },
  "result": [
    {
      "id": 103,
      "modificationCounter": 1,
      "revision": null,
      "waiterId": null,
      "number": 3,
      "state": "FREE"
    },
    {
      "id": 104,
      "modificationCounter": 1,
      "revision": null,
      "waiterId": null,
      "number": 4,
      "state": "FREE"
    }
  ]
}
```

## Query Meta-Parameters

Queries can have meta-parameters and the OASP4J currently provides support for `timeout`. OASP4J provides the method `applyCriteria` in `AbstractGenericDao` that applies meta-parameters to a query based on `SearchCriteriaTo`. If you already use the pagination support (see above), you do not need to call `applyCriteria` manually, as it is called internally by `findPaginated`.

## Relationships

[[getting-started-data-access-layer\_n:1-and 1:1 relationships]] ===== n:1 and 1:1 Relationships Entities often do not exist independently but are in some relation to each other. For example, for every period of time one of the `StaffMember`'s of the restaurant example has worked, which is represented by the class `WorkingTime`, there is a relationship to this `StaffMember`.

The following listing shows how this can be modeled using JPA:

```
...
@Entity
public class WorkingTime {
  ...
  private StaffMember staffMember;
  ...
  @ManyToOne
  @JoinColumn(name="STAFFMEMBER")
```

```
public StaffMember getStaffMember() {
    return staffMember;
}

public void setStaffMember(StaffMember staffMember) {
    this.staffMember = staffMember;
}
```

To represent the relationship, an attribute of the type of the corresponding entity class that is referenced has been introduced. The relationship is a n:1 relationship, because every *WorkingTime* belongs to exactly one *StaffMember*, but a *StaffMember* usually worked more often than once. This is why the `@ManyToOne` annotation is used here. For 1:1 relationships the `@OneToOne` annotation can be used which works basically the same way. To be able to save information about the relation in the database, an additional column in the corresponding table of *WorkingTime* is needed which contains the primary key of the referenced *StaffMember*. With the `name` element of the `@JoinColumn` annotation it is possible to specify the name of this column.

[[getting-started-data-access-layer\_1:n-and n:m relationships]] ===== 1:n and n:m Relationships The relationship of the example listed above is currently an unidirectional one, as there is a getter method for retrieving the *StaffMember* from the *WorkingTime* object, but not vice versa.

To make it a bidirectional one, the following code has to be added to *StaffMember*:

```
private Set<WorkingTimes> workingTimes;

@OneToMany(mappedBy="staffMember")
public Set<WorkingTime> getWorkingTimes() {
    return workingTimes;
}

public void setWorkingTimes(Set<WorkingTime> workingTimes) {
    this.workingTimes = workingTimes;
}
```

To make the relationship bidirectional, the tables in the database do not have to be changed. Instead the column that corresponds to the attribute *staffMember* in class *WorkingTime* is used, which is specified by the `mappedBy` element of the `@OneToMany` annotation. Hibernate will search for corresponding *WorkingTime* objects automatically when a *StaffMember* is loaded.

The problem with bidirectional relationships is that if a *WorkingTime* object is added to the set or list *workingTimes* in *StaffMember*, this does not have any effect in the database unless the *staffMember* attribute of that *WorkingTime* object is set. That is why the OASp4J advises not to use bidirectional relationships but to use queries instead. How to do this is shown [here](#). If a bidirectional relationship should be used nevertheless, appropriate *add* and *remove* methods must be used.

For 1:n and n:m relations, the OASp4J demands that (unordered) Sets and no other collection types are used, as shown in the listing above. The only exception is whenever an ordering is really needed, (sorted) lists can be used. For example, if *WorkingTime* objects should be sorted by their start time, this could be done like this:

```
private List<WorkingTimes> workingTimes;

@OneToMany(mappedBy = "staffMember")
@OrderBy("startTime asc")
public List<WorkingTime> getWorkingTimes() {
    return workingTimes;
}
```

```
public void setWorkingTimes(List<WorkingTime> workingTimes) {  
    this.workingTimes = workingTimes;  
}
```

The value of the `@OrderBy` annotation consists of an attribute name of the class followed by `asc` (ascending) or `desc` (descending).

To store information about a n:m relationship, a separate table has to be used, as one column cannot store several values (at least if the database schema is in first normal form). In the example application, in the case of the `Bill` and the `orderPositions` the relation between them could be modeled as follows:

```
private List<OrderPositionEntity> orderPositions;  
  
@ManyToMany(fetch = FetchType.EAGER)  
@JoinTable(name = "Bill_OrderPosition", joinColumns = { @JoinColumn(name = "Bill_id") })  
public List<OrderPositionEntity> getOrderPositions() {  
  
    return this.orderPositions;  
}  
  
public void setOrderPositions(List<OrderPositionEntity> orderPositions) {  
  
    this.orderPositions = orderPositions;  
}
```

Information about the relation is stored in a table called `BILL_ORDERPOSITION` that has to have two columns, one for referencing the Bill (`bill_id`), the other one for referencing the Order (`orderpositions_id`). Note that the `@JoinTable` annotation is not needed in this case because a separate table is the default solution here (same for n:m relations) unless there is a `mappedBy` element specified.

For 1:n relationships this solution has the disadvantage that more joins (in the database system) are needed to get a Bill with all the Order's it refers to. This might have a negative impact on performance so that the solution to store a reference to the Bill row/entity in the Order's table is probably the better solution in most cases.

Note that bidirectional n:m relationships are not allowed for applications based on the OASP4J. Instead a third entity has to be introduced, which "represents" the relationship (it has two n:1 relationships).

[[getting-started-data-access-layer-eager-vs. lazy loading]] ===== Eager vs. Lazy Loading Using JPA/ Hibernate it is possible to use either lazy or eager loading. Eager loading means that for entities retrieved from the database, other entities that are referenced by these entities are also retrieved, whereas lazy loading means that this is only done when they are actually needed, i.e. when the corresponding getter method is invoked.

Application based on the OASP4J must use lazy loading per default. Projects generated with the project generator are already configured so that this is actually the case.

For some entities it might be beneficial if eager loading is used. For example if every time a `Bill` is processed, the `Order` entities it refers to are needed, eager loading can be used as shown in the following listing:

```
@OneToMany(fetch = FetchType.EAGER)  
@JoinTable  
public Set<Order> getOrders() {  
    return orders;  
}
```

This can be done with all four types of relationships (annotations: `@OneToOne`, `@ManyToOne`, `@OneToMany`, `@ManyToOne`).

## Cascading Relationships

It is not only possible to specify what happens if an entity is loaded that has some relationship to other entities (see above), but also if an entity is for example persisted or deleted. By default, nothing is done in these situations. This can be changed by using the `cascade` element of the annotation that specifies the relation type (`@OneToOne`, `@ManyToOne`, `@OneToMany`, `@ManyToOne`). For example, if a `StaffMember` is persisted, all its `WorkingTime`'s should be persisted and if the same applies for deletions (and some other situations, for example if an entity is reloaded from the database, which can be done using the `refresh(Object)` method of an `EntityManager`), this can be done as shown in the following listing:

```
@OneToOne(mappedBy = "staffMember", cascade=CascadeType.ALL)
public Set<WorkingTime> getWorkingTime() {
    return workingTime;
}
```

There are several `CascadeTypes`, e.g. to specify that a "cascading behavior" should only be used if an entity is persisted (`CascadeType.PERSIST`) or deleted (`CascadeType.REMOVE`), see [here](#) for more information.

## Embeddable

An embeddable Object is a way to implement [relationships](#) between [entities](#), but with a mapping in which both entities are in the same database table. If these entities are often needed together, this is a good way to speed up database operations, as only one access to a table is needed to retrieve both entities.

Suppose the restaurant example application has to be extended in a way that it is possible to store information about the addresses of `StaffMember`'s, this can be done with a new `Address` class:

```
...
@Embeddable
public class Address {

    private String street;

    private String number;

    private Integer zipCode;

    private String city;

    @Column(name="STREETNUMBER")
    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    ... // other getter and setter methods, equals, hashCode
}
```

This class looks a bit like an entity class, apart from the fact that the `@Embeddable` annotation is used instead of the `@Entity` annotation and no primary key is needed here. In addition to that the methods `equals(Object)` and `hashCode()` need to be implemented as this is required by Hibernate (it is not required for entities because they can be unambiguously identified by their primary key). For some hints on how to implement the `hashCode()` method please have a look [here](#).

Using the address in the `StaffMember` entity class can be done as shown in the following listing:

```
...
@Entity
@Table(name = "StaffMember")
public class StaffMemberEntity implements StaffMember {

    ...
    private Address address;
    ...

    @Embedded
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

The `@Embedded` annotation needs to be used for embedded attributes. Note that if in all columns in the `StaffMember`'s table that belong to the `Address` embeddable the values are null, the `Address` will be null when retrieving the `StaffMember` entity from the database. This has to be considered when implementing the application core to avoid `NullPointerException`'s.

Moreover, if the database tables are created automatically by Hibernate and a primitive data type is used in the embeddable (in the example this would be the case if `int` is used instead of `Integer` as data type for the `zipCode`), there will be a *not null* constraint on the corresponding column (reason: a primitive data type can never be null in java, so hibernate always introduces a *not null* constraint). This constraint would be violated if one tries to insert a `StaffMember` without an `Address` object (this might be considered as a bug in Hibernate).

Another way to realize the one table mapping are Hibernate UserType's, as described [here](#).

## Inheritance

Just like normal java classes, [entity](#) classes can inherit from others. The only difference is that you need to specify how to map a subtype hierarchy to database tables.

The [Java Persistence API \(JPA\)](#) offers three ways to do this:

- [One table per hierarchy](#): This table contains all columns needed to store all types of entities in the hierarchy. If a column is not needed for an entity because of its type, there is a null value in this column. An additional column is introduced, which denotes the type of the entity (called "`dtype`" which is of type `varchar` and stores the class name).
- [One table per concrete class](#). For each concrete entity class there is a table in the database that can store such an entity with all its attributes. An entity is only saved in the table corresponding to its most concrete type. To get all entities of a type that has subtypes, joins are needed.
- [One table per subclass](#): In this case there is a table for every entity class (this includes abstract classes), which contains all columns needed to store an entity of that class apart from those that are already included in the table of the supertype. Additionally there is a primary key column in every table. To get an entity of a class that is a subclass of another one, joins are needed.

Every of the three approaches has its advantages and drawbacks, which are discussed in detail [here](#). In most cases, the first one should be used, because it is usually the fastest way to do the mapping,

as no joins are needed when retrieving entities and persisting a new entity or updating one only affects one table. Moreover it is rather simple and easy to understand. One major disadvantage is that the first approach could lead to a table with a lot of null values, which might have a negative impact on the database size.

The following listings show how to create a class hierarchy among entity classes for the class *FoodDrink* and its subclass *Drink*:

```
...

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class FoodDrink {

    private long id;

    private String description;

    private byte[] picture;

    private long version;

    @Id
    @Column(name = "ID")
    @GeneratedValue(generator = "SEQ_GEN")
    @SequenceGenerator(name = "SEQ_GEN", sequenceName = "SEQ_FOODDRINK")
    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }

    ...
}

...

@Entity
public class Drink extends FoodDrink {

    private boolean alcoholic;

    public boolean isAlcoholic() {
        return alcoholic;
    }

    public void setAlcoholic(boolean alcoholic) {
        this.alcoholic = alcoholic;
    }
}
```

To specify how to map the class hierarchy, the `@Inheritance` annotation is used. Its element `strategy` defines which type of mapping is used and can have the following values:

- `InheritanceType.SINGLE_TABLE` (= one table per hierarchy).
- `InheritanceType.TABLE_PER_CLASS` (= one table per concrete class).
- `InheritanceType.JOINED` (= one table per subclass, joined tables).

As a best practice we advise you to avoid deep class hierarchies among entity classes (unless they reduce complexity).

## Concurrency Control

The concurrency control defines the way concurrent access to the same data of a database is handled. When several users (or threads of application servers) are concurrently accessing a database, anomalies may happen, e.g. a transaction is able to see changes from another transaction although that one did not yet commit these changes. Most of these anomalies are automatically prevented by the database system, depending on the [\*isolation level\*](#) (property `hibernate.connection.isolation` in the `jpa.xml`, see [here](#)).

A remaining anomaly is when two stakeholders concurrently access a record, do some changes and write them back to the database. The JPA addresses this with different locking strategies (see [here](#) or [here](#)).

As a best practice we are using optimistic locking for regular end-user [services](#) (OLTP) and pessimistic locking for [batches](#).

### Optimistic Locking

The class `io.oasp.module.jpa.persistence.api.AbstractPersistenceEntity` already provides optimistic locking via a `modificationCounter` with the `@Version` annotation. Therefore JPA takes care of optimistic locking for you. When entities are transferred to clients, modified and sent back for update you need to ensure the `modificationCounter` is part of the game. If you follow our guides about [transfer-objects](#) and [services](#) this will also work out of the box. You only have to care about two things:

- How to deal with optimistic locking in [relationships](#)?

Assume an entity `A` contains a collection of `B` entities. Should there be a locking conflict if one user modifies an instance of `A` while another user in parallel modifies an instance of `B` that is contained in the other instance? To take influence besides placing collections take a look at [GenericDao.forceIncrementModificationCounter](#).

- What should happen in the UI if an `OptimisticLockException` occurred? According to KISS our recommendation is that the user gets an error displayed that tells him to do his change again on the recent data. Try to design your system and the work processing in a way to keep such conflicts rare.

### Pessimistic Locking

For back-end [services](#) and especially for [batches](#) optimistic locking is not suitable. A human user shall not cause a large batch process to fail because he was editing the same entity. Therefore such use-cases use pessimistic locking what gives them a kind of priority over the human users. In your [DAO](#) implementation you can provide methods that do pessimistic locking via [EntityManager](#) operations that take a [LockModeType](#). Here is a simple example:

```
getEntityManager().lock(entity, LockModeType.READ);
```

When using the `lock(Object, LockModeType)` method with `LockModeType.READ`, Hibernate will issue a `select ... for update`. This means that no one else can update the entity (see [here](#) for more information on the statement). If `LockModeType.WRITE` is specified, Hibernate issues a `select ... for update nowait` instead, which has the same meaning as the statement above, but if there is already a lock, the program will not wait for this lock to be released. Instead, an exception is raised. Use one of the types if you want to modify the entity later on, for read only access no lock is required.

As you might have noticed, the behavior of Hibernate deviates from what one would expect by looking at the `LockModeType` (especially `LockModeType.READ` should not cause a `select ... for update` to be issued). The framework actually deviates from what is [specified](#) in the JPA for unknown reasons.

## Database Auditing

See [auditing guide](#).

[[getting-started-data-access-layer\_testing-entities and daos]] === Testing Entities and DAOs See [testing guide](#).

[[getting-started-data-access-layer\_summary-of-principles]] === Summary of principles We strongly recommend these principles:

- Use the JPA where ever possible and use vendor (Hibernate) specific features only for situations when JPA does not provide a solution. In the latter case consider first if you really need the feature.
- Create your entities as simple POJOs and use JPA to annotate the getters in order to define the mapping.
- Keep your entities simple and avoid putting advanced logic into entity methods.

## 2.16.2 Database Configuration

The [configuration](#) for Spring and Hibernate is already provided by OASP4J in our sample application and the application template. So you only need to worry about a few things to customize.

[[getting-started-data-access-layer\_database-system and access]] === Database System and Access Obviously you need to configure which type of database you want to use as well as the location and credentials to access it. To configure a new database we must follow next steps

### Add dependencies

Add Dependency for database in pom.xml file. For e.g if we are configuring mysql database in our application we need to add below dependency in pom.xml file.

MySQL:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

**Note:** this driver should NOT be used in a production environment because of license issues. See down for an alternative.

[[getting-started-data-access-layer\_create-a spring profile]] ===== Create a Spring Profile

Spring Profile provides a way to segregate parts of your application configuration and make it only available in certain environments. In our case, we're going to create a profile to our database configuration.

Create a file in `src/resources/config` named `application-{PROFILE_NAME}.properties`. The name of the profile isn't important, for this example we named it `application-mysql.properties`.

[[getting-started-data-access-layer\_set-the database configuration]] ===== Set the database configuration

Add database configuration in `application-{PROFILE_NAME}.properties` file. For e.g

MySQL:

```
spring.jpa.database=mysql
spring.datasource.url=jdbc:mysql://localhost:3306/restaurant?
user=root&password=root&connectTimeout=10000&socketTimeout=10000&autoReconnect=true
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
flyway.locations=classpath:db/migration,classpath:db/migration/mysql
```

[[getting-started-data-access-layer\_activate-the new profile]] ===== Activate the new profile

To activate new profile update *src/main/resources/application.properties* file. Set *spring.profiles.active=profile-name*. In our example profile-name is mysql.

MySQL:

```
spring.profiles.active=mysql
```

[[getting-started-data-access-layer\_update-database script files]] ===== Update database script files

Devonfw has [flyway](#) configured. Flyway is an open-source database migration tool. It strongly favors simplicity and convention over configuration. Flyway will search for script files for corresponding database. It will parse the script files and create or update corresponding tables in a database.

Generally, DDL Script file is present at location *db/migration/database/1.0*. For e.g *db/migration/mysql/1.0* And other script files are present at location *db/migration*. Make sure script files are error free. We can set customized location for migration scripts. We need to add *flyway.locations* property in *application.properties*. For example

```
flyway.locations=classpath:db/migration,classpath:db/migration/mysql
```

Here we can mention classpath or filesystems path.

You can see more examples of database configurations [here](#)

## Database Migration

See [database migration guide](#).

### 2.16.3 Security

#### SQL injection

A common [security](#) threat is [SQL-injection](#). Never build queries with string concatenation or your code might be vulnerable as in the following example:

```
String query = "Select op from OrderPosition op where op.comment = " + userInput;
return getEntityManager().createQuery(query).getResultList();
```

Via the parameter *userInput* an attacker can inject SQL (JPQL) and execute arbitrary statements in the database causing extreme damage. In order to prevent such injections you have to strictly follow our rules for [queries](#): Use named queries for static queries and QueryDSL for dynamic queries. Please also consult the [SQL Injection Prevention Cheat Sheet](#).

[[getting-started-data-access-layer\_limited-permissions for application]] === Limited Permissions for Application We suggest that you operate your application with a database user that has limited

---

permissions so he can not modify the SQL schema (e.g. drop tables). For initializing the schema (DDL) or to do schema migrations use a separate user that is not used by the application itself.

## 2.17 CRUD operations and DAO implementation

[[getting-started-crud-operations\_create-crud functionality for an entity]] == Create CRUD functionality for an entity

In this tutorial, we are going to create an entity for the application and provide services for the typical Create, Read, Update and Delete operations for that entity.

If you want to create the application from scratch:

- Launch the *console.bat* script.
- Go to a desired directory and use the Maven command

```
mvn -DarchetypeVersion=2.3.0 -DarchetypeGroupId=io.oasp.java.templates -DarchetypeArtifactId=oasp4j-  
template-server archetype:generate -DgroupId=com.capgemini.devonfw.application -DartifactId=tutorial -  
Dversion=0.1-SNAPSHOT -Dpackage=devonfw.tutorial
```

- Open Eclipse and Import the new *tutorial* project as *Existing Maven Project*

If you want to know more about how to create a new application you can visit the [Create New Application](#) section.

Before continue it is important to keep in mind the packaging convention that Devonfw proposes. Devonfw uses a strict packaging convention to map technical layers and business components to the code. Devonfw uses the following Java-Package schema:

```
<basepackage>.<component>.<layer>.<scope>[.<detail>]*
```

In our example application we find the different classes in this packages:

- Entity and DAO: devonfw.tutorial.tablemanagement.dataaccess.api[.<detail>]
- Logic: devonfw.tutorial.tablemanagement.logic[.<detail>]
- Services: devonfw.tutorial.tablemanagement.services[.<detail>]

This convention is based on the OASP4J conventions, which you can consult in the [OASP4J Coding conventions documentation](#)

[[getting-started-crud-operations\_step-1: add the database schema]] ===== Step 1: Add the database schema As first step we are going to add the database schema to our database.

In the script `resources/db/migration/V0001__Create_schema.sql` we add:

```
CREATE CACHED TABLE PUBLIC.RESTAURANTTABLE(  
    id BIGINT NOT NULL,  
    modificationCounter INTEGER NOT NULL,  
    number BIGINT NOT NULL CHECK (NUMBER >= 0),  
    state INTEGER,  
    waiter_id BIGINT  
) ;
```

And in the same path, we are going to create a new file to add the default data to the `RestaurantTable` created. We create `V0002__Master_data.sql` file.

```
INSERT INTO RESTAURANTTABLE (id, modificationCounter, number, state) VALUES (101, 1, 1, 2);
INSERT INTO RESTAURANTTABLE (id, modificationCounter, number, state) VALUES (102, 1, 2, 0);
INSERT INTO RESTAURANTTABLE (id, modificationCounter, number, state) VALUES (103, 1, 3, 0);
INSERT INTO RESTAURANTTABLE (id, modificationCounter, number, state) VALUES (104, 1, 4, 0);
INSERT INTO RESTAURANTTABLE (id, modificationCounter, number, state) VALUES (105, 1, 5, 0);
```

[[getting-started-crud-operations\_step-2: create the jpa entity]] ===== Step 2: Create the JPA entity We are going to create a Table entity and its related interface (that will be reused between all the objects involved with tables in the different layers).

### Note

This tutorial uses the same use-cases and scenario as the OASP4J sample application: Modelling a restaurant.

Do not confuse Table with a DB-table. In this context, we mean a table where the guests of the restaurant are seated.

Create the devonfw.tutorial.tablemanagement.common.api package in the tutorial-server-core project, by right-clicking on the project, and selecting *New > Package*.

Create the interface Table inside the devonfw.tutorial.tablemanagement.common.api package (you can do it by right-clicking on the package and selecting *New > Interface*), and copy & paste the following code:

```
package devonfw.tutorial.tablemanagement.common.api;

import devonfw.tutorial.general.common.api.ApplicationEntity;
import devonfw.tutorial.tablemanagement.common.api.datatype.TableState;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

/**
 * This is the interface for a table of the restaurant. It has a unique {@link #getNumber()} number} can be
 * {@link TableState#isReserved()} reserved}, {@link TableState#isOccupied()} occupied} and may have a
 * {@link #getWaiterId()} waiter} assigned.
 */
public interface Table extends ApplicationEntity {

    /**
     * @return the unique table number.
     */
    @NotNull
    @Min(0)
    Long getNumber();

    /**
     * @param number is the new {@link #getNumber()} number}.
     */
    void setNumber(Long number);

    /**
     * @return the current {@link TableState state} of this {@link Table}.
     */
    TableState getState();

    /**
     * @param state is the new {@link #getState()} state}.
     */
    void setState(TableState state);

    /**

```

```
 * @return the {@link devonfw.tutorial.staffmanagement.common.api.StaffMember#getId() ID} of the
waiter
 *      currently responsible for this table.
 */
Long getWaiterId();

/**
 * Sets the field 'waiterId'.
 *
 * @param waiterId New value for waiterId
 */
void setWaiterId(Long waiterId);

}
```

### Note

You may have compilation errors related to *TableState* that is not yet implemented. We will take care of that in the next step.

As you can see, Table extends `ApplicationEntity` class, as is recommended for standard mutable entities of an application. This class provides the necessary methods for a mutable entity (ID getter and setter basically).

In the above Table class, we save the state of the table by using a `TableState` enum, which we will create now:

Create the package `devonfw.tutorial.tablemanagement.common.api.datatype`, and inside this package, create a new class (actually an enum) called `TableState` and copy & paste the code below (as mentioned before you can use the right-click option over the `datatype` package and select *New > Enum*).

```
package devonfw.tutorial.tablemanagement.common.api.datatype;

/**
 * Represents the {@link devonfw.tutorial.tablemanagement.common.api.Table# getState() state} of a
 * {@link devonfw.tutorial.tablemanagement.common.api.Table}.
 */
public enum TableState {
    /** The state of a free {@link devonfw.tutorial.tablemanagement.common.api.Table}. */
    FREE,

    /** The state of a reserved {@link devonfw.tutorial.tablemanagement.common.api.Table}. */
    RESERVED,

    /** The state of a occupied {@link devonfw.tutorial.tablemanagement.common.api.Table}. */
    OCCUPIED,

    /**
     * @return {@code true} if {@link #FREE}, {@code false} otherwise.
     */
    public boolean isFree() {
        return (this == FREE);
    }

    /**
     * @return {@code true} if {@link #RESERVED}, {@code false} otherwise.
     */
    public boolean isReserved() {
        return (this == RESERVED);
    }
}
```

```
/**  
 * @return {@code true} if {@link #OCCUPIED}, {@code false} otherwise.  
 */  
public boolean isOccupied() {  
  
    return (this == OCCUPIED);  
}  
}
```

## Note

It is possible that Eclipse removed the import of the `TableState` enum in the `Table` interface, if you saved the file before creating the `TableState` class.

If Eclipse shows errors still, after you've created the `TableState` enum, open the `Table` interface and press **Ctrl-Shift-O** to automatically fix the 'class' imports.

Finally, we should create the entity implementation. Create the package `devonfw.tutorial.tablemanagement.dataaccess.api`, create the class `TableEntity` inside it and paste the following code:

```
package devonfw.tutorial.tablemanagement.dataaccess.api;  
  
import devonfw.tutorial.general.dataaccess.api.ApplicationPersistenceEntity;  
import devonfw.tutorial.tablemanagement.common.api.Table;  
import devonfw.tutorial.tablemanagement.common.api.datatype.TableState;  
  
import javax.persistence.Column;  
import javax.persistence.Entity;  
  
/**  
 * {@link ApplicationPersistenceEntity} Entity representing a {@link Table} of the restaurant. A table  
 * has a unique  
 * {@link #getNumber()} number can be {@link TableState#isReserved()} reserved}, {@link  
 * TableState#isOccupied()} occupied}  
 * and may have a {@link devonfw.tutorial.staffmanagement.dataaccess.api.StaffMemberEntity waiter}  
 * assigned.  
 */  
@Entity  
// Table is a reserved word in SQL/RDBMS and can not be used as table name  
@javax.persistence.Table(name = "RestaurantTable")  
public class TableEntity extends ApplicationPersistenceEntity implements Table {  
  
    private static final long serialVersionUID = 1L;  
  
    private Long number;  
  
    private Long waiterId;  
  
    private TableState state;  
  
    @Override  
    @Column(unique = true)  
    public Long getNumber() {  
  
        return this.number;  
    }  
  
    @Override  
    public void setNumber(Long number) {  
  
        this.number = number;  
    }  
  
    @Override  
    @Column(name = "waiter_id")
```

```
public Long getWaiterId() {
    return this.waiterId;
}

@Override
public void setWaiterId(Long waiterId) {
    this.waiterId = waiterId;
}

@Override
public TableState getState() {
    return this.state;
}

@Override
public void setState(TableState state) {
    this.state = state;
}
```

## 2.17.1 Validation

We want tables to never have negative numbers, so we are going to add a validation to our `TableEntity`. Change the definition of the `getNumber` method of the `TableEntity` class as follows:

```
@Min(0)
@Column(unique = true)
public Long getNumber() {
    return this.number;
}
```

### Note

You may need to solve the import of the `@Min` annotation by right clicking over the annotation and selecting `import javax.validation.constraints.Min`. You can read more about validation in [the OASp4J guide about validation](#)

[[getting-started-crud-operations\_step-3: create persistence layer]] ===== Step 3: Create persistence layer

Data Access Objects (DAOs) are part of the persistence layer. They are responsible for a specific entity and should be named as `<Entity>Dao[Impl]`. The DAO offers the so called CRUD-functionalities (create, retrieve, update, delete) for the corresponding entity. Additionally a DAO may offer advanced operations such as search or locking methods.

For each DAO there is an interface named `<Entity>Dao` that defines the API. For CRUD support and common naming methods we derive it from the interface `devonfw.tutorial.general.dataaccess.api.dao.ApplicationDao`, which was automatically generated while using the OASp4J archetype to generate your application

### Note

For the sake of simplicity, in the rest of this tutorial, we will no longer specifically tell you to create java packages for new java classes.

Instead, we ask you to pay attention to the first line of each new java file, and create, if necessary, the class' package.

Create the following DAO interface for our Table entity:

### TableDao.java.

```
package devonfw.tutorial.tablemanagement.dataaccess.api.dao;

import devonfw.tutorial.general.dataaccess.api.dao.ApplicationDao;
import devonfw.tutorial.tablemanagement.dataaccess.api.TableEntity;
import io.oasp.module.jpa.dataaccess.api.MasterDataDao;

import java.util.List;

/**
 * {@link ApplicationDao Data Access Object} for {@link TableEntity} entity.
 */
public interface TableDao extends ApplicationDao<TableEntity>, MasterDataDao<TableEntity> {

    /**
     * Returns a list of free restaurant tables.
     *
     * @return {@link List} of free restaurant {@link TableEntity}s
     */
    List<TableEntity> getFreeTables();
}
```

## 2.17.2 Define queries

Before we proceed to the implementation of this DAO interface, we will create the SQL query.

OASP4J advises to specify all queries in one mapping file called *orm.xml* located in *src/main/resources/META-INF*. So we are going to create a query to return all free tables that we will use in *TableDaoImpl*.

### src/main/resources/META-INF/orm.xml.

```
<!--?xml version="1.0" encoding="UTF-8"?-->
<entity-mappings version="1.0" xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemalocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">

    <named-query name="get.free.tables">
        <query><![CDATA[ SELECT t FROM TableEntity t WHERE t.state = devonfw.tutorial.tablemanagement.common.api.datatype.TableState.FREE ]]></query>
    </named-query>

</entity-mappings>
```

To avoid redundant occurrences of the query name we are going to use a constants class where we are going to define the constants for each named query:

### NamedQueries.java.

```
package devonfw.tutorial.general.common.api.constants;

/**
 * Constants of the named queries defined in ``NamedQueries.xml``.
 */
public abstract class NamedQueries {

    // put your query names from NamedQueries.xml as constants here
    /** @see devonfw.tutorial.tablemanagement.dataaccess.impl.dao.TableDaoImpl#getFreeTables() */
    public static final String GET_FREE_TABLES = "get.free.tables";
```

{}

Note that changing the name of the java constant can be done easily with refactoring (right-clicking over the property and *Refactor > Rename*). Further you can trace where the query is used by searching the references of the constant.

[[getting-started-crud-operations\_implementation-of dao interface]] ===== Implementation of DAO interface

Implementing a DAO is quite simple. We create a class named `<Entity>DaoImpl` that extends `ApplicationMasterDataDaoImpl` class and implements our DAO interface.

This is the DAO implementation for our `TableDao` interface:

### TableDaoImpl.java.

```
package devonfw.tutorial.tablemanagement.dataaccess.impl.dao;

import java.util.List;

import javax.inject.Named;
import javax.persistence.Query;

import devonfw.tutorial.general.common.api.constants.NamedQueries;
import devonfw.tutorial.general.dataaccess.base.dao.ApplicationMasterDataDaoImpl;
import devonfw.tutorial.tablemanagement.dataaccess.api.TableEntity;
import devonfw.tutorial.tablemanagement.dataaccess.api.dao.TableDao;

/**
 * Implementation of {@link TableDao}.
 */
@Named
public class TableDaoImpl extends ApplicationMasterDataDaoImpl<TableEntity> implements TableDao {

    /**
     * The constructor.
     */
    public TableDaoImpl() {
        super();
    }

    @Override
    public Class<TableEntity> getEntityClass() {
        return TableEntity.class;
    }

    @Override
    public List<TableEntity> getFreeTables() {
        Query query = getEntityManager().createNamedQuery(NamedQueries.GET_FREE_TABLES, TableEntity.class);
        return query.getResultList();
    }
}
```

As you can see `ApplicationMasterDataDaoImpl` already implements the CRUD operations so you only have to implement the additional methods that you have declared in your `<entity>Dao` interface.

[[getting-started-crud-operations\_step-4: business logic]] ===== Step 4: Business logic

The business logic of our application is defined in the logic layer, as proposed by the OASP4J Guide.

The logic layer also maps entities from the dataaccess layer to/from transfer objects, so we do not expose internal details of the applications implementation to higher layers.

In Devonfw applications, there are several different types of *Transfer Objects* (short TO). One is the *Entity Transfer Object* (ETO) used to transfer a representation of an Entity.

As a first step, we will define an ETO for the Table entity, to be used in the interface of our logic layer.

Create the following file:

### TableEto.java.

```
package devonfw.tutorial.tablemanagement.logic.api.to;

import devonfw.tutorial.general.common.api.to.AbstractEto;
import devonfw.tutorial.tablemanagement.common.api.Table;
import devonfw.tutorial.tablemanagement.common.api.datatype.TableState;

import javax.validation.constraints.Max;

/**
 * {@link AbstractEto ETO} for {@link Table}.
 */
public class TableEto extends AbstractEto implements Table {

    private static final long serialVersionUID = 1L;

    private Long waiterId;

    @Max(value = 1000)
    private Long number;

    private TableState state;

    /**
     * The constructor.
     */
    public TableEto() {

        super();
    }

    @Override
    public Long getNumber() {

        return this.number;
    }

    @Override
    public void setNumber(Long number) {

        this.number = number;
    }

    @Override
    public Long getWaiterId() {

        return this.waiterId;
    }

    @Override
    public void setWaiterId(Long waiterId) {

        this.waiterId = waiterId;
    }

    @Override
    public TableState getState() {

        return this.state;
    }
}
```

```
    @Override
    public void setState(TableState state) {
        this.state = state;
    }

    @Override
    public int hashCode() {

        final int prime = 31;
        int result = super.hashCode();
        result = prime * result + ((this.state == null) ? 0 : this.state.hashCode());
        result = prime * result + ((this.waiterId == null) ? 0 : this.waiterId.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {

        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        if (!super.equals(obj)) {
            return false;
        }
        TableEto other = (TableEto) obj;
        if (this.state != other.state) {
            return false;
        }
        if (this.waiterId == null) {
            if (other.waiterId != null) {
                return false;
            }
        } else if (!this.waiterId.equals(other.waiterId)) {
            return false;
        }
        return true;
    }
}
```

In Devonfw, we define CRUD logic into a <Entity>management class. So we are going to create our Tablemanagement interface and implementation:

### Tablemanagement.java.

```
package devonfw.tutorial.tablemanagement.logic.api;

import devonfw.tutorial.tablemanagement.logic.api.to.TableEto;

import java.util.List;

import javax.validation.Valid;

/**
 * Interface for TableManagement component.
 *
 */
public interface Tablemanagement {

    /**
     * Returns a restaurant table by its id 'id'.
     *
     * @param id The id 'id' of the restaurant table.
     */
}
```

```
 * @return The restaurant {@link TableEto} with id 'id'
 */
TableEto findTable(Long id);

/**
 * Returns a list of all existing restaurant tables.
 *
 * @return {@link List} of all existing restaurant {@link TableEto}s
 */
List<TableEto> findAllTables();

/**
 * Returns a list of all existing free restaurant tables.
 *
 * @return {@link List} of all existing free restaurant {@link TableEto}s
 */
List<TableEto> findFreeTables();

/**
 * Deletes a restaurant table from the database by its id 'id'.
 *
 * @param tableId Id of the restaurant table to delete
 */
void deleteTable(Long tableId);

/**
 * Creates a new restaurant table and store it in the database.
 *
 * @param table the {@link TableEto} to create.
 * @return the new {@link TableEto} that has been saved with ID and version.
 */
TableEto saveTable(@Valid TableEto table);
}
```

### TablemanagementImpl.java.

```
package devonfw.tutorial.tablemanagement.logic.impl;

import devonfw.tutorial.general.common.api.constants.PermissionConstants;
import devonfw.tutorial.general.common.api.exception.IllegalEntityStateException;
import devonfw.tutorial.general.logic.base.AbstractComponentFacade;
import devonfw.tutorial.tablemanagement.common.api.datatype.TableState;
import devonfw.tutorial.tablemanagement.dataaccess.api.TableEntity;
import devonfw.tutorial.tablemanagement.dataaccess.api.dao.TableDao;
import devonfw.tutorial.tablemanagement.logic.api.Tablemanagement;
import devonfw.tutorial.tablemanagement.logic.api.to.TableEto;

import java.util.List;
import java.util.Objects;

import javax.annotation.security.RolesAllowed;
import javax.inject.Inject;
import javax.inject.Named;
import javax.validation.Valid;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Implementation of {@link Tablemanagement}.
 */
@Named
public class TablemanagementImpl extends AbstractComponentFacade implements Tablemanagement {

    /** Logger instance. */
    private static final Logger LOG = LoggerFactory.getLogger(TablemanagementImpl.class);

    /** @see #getTableDao() */
    private TableDao tableDao;
```

```
/**  
 * The constructor.  
 */  
public TablemanagementImpl() {  
    super();  
}  
  
@Override  
@RolesAllowed(PermissionConstants.FIND_TABLE)  
public TableEto findTable(Long id) {  
  
    LOG.debug("Get table with id '" + id + "' from database.");  
    return getBeanMapper().map(getTableDao().findOne(id), TableEto.class);  
}  
  
@Override  
@RolesAllowed(PermissionConstants.FIND_TABLE)  
public List<TableEto> findAllTables() {  
  
    LOG.debug("Get all restaurant tables from database.");  
    List<TableEntity> tables = getTableDao().findAll();  
    return getBeanMapper().mapList(tables, TableEto.class);  
}  
  
@Override  
@RolesAllowed(PermissionConstants.FIND_TABLE)  
public List<TableEto> findFreeTables() {  
  
    LOG.debug("Get all free restaurant tables from database.");  
  
    List<TableEntity> tables = getTableDao().getFreeTables();  
    return getBeanMapper().mapList(tables, TableEto.class);  
}  
  
@Override  
@RolesAllowed(PermissionConstants.DELETE_TABLE)  
public void deleteTable(Long tableId) {  
  
    TableEntity table = getTableDao().find(tableId);  
  
    if (!table.getState().isFree()) {  
        throw new IllegalEntityStateException(table, table.getState());  
    }  
  
    getTableDao().delete(table);  
}  
  
@Override  
@RolesAllowed(PermissionConstants.SAVE_TABLE)  
public TableEto saveTable(@Valid TableEto table) {  
  
    Objects.requireNonNull(table, "table");  
  
    TableEntity tableEntity = getBeanMapper().map(table, TableEntity.class);  
    // initialize  
    if (tableEntity.getState() == null) {  
        tableEntity.setState(TableState.FREE);  
    }  
  
    getTableDao().save(tableEntity);  
    LOG.debug("Table with id '{}' has been created.", tableEntity.getId());  
    return getBeanMapper().map(tableEntity, TableEto.class);  
}  
  
/**  
 * @return the {@link TableDao} instance.  
 */  
public TableDao getTableDao() {  
  
    return this.tableDao;  
}
```

```
/**  
 * @param tableDao the {@link TableDao} to {@link Inject}.  
 */  
@Inject  
public void setTableDao(TableDao tableDao) {  
  
    this.tableDao = tableDao;  
}
```

## Note

You may have problems with the *PermissionConstants* properties because are not implemented yet. We will do that in the next step.

At this point we have defined all the necessary classes in the logic layer, so we have our API ready, with the exception of finishing its security aspect.

[[getting-started-crud-operations\_secure-the application]] ===== Secure the application

OASP4J proposes role-based authorization to cope with the authorization of executing use cases of an application. OASP4J use the *JSR250* annotations, mainly *@RolesAllowed*, as you have seen, for authorizing method calls against the permissions defined in the annotation body.

So, finally, we have to create a class to declare the actual roles we use as values for the *@RolesAllowed* annotation:

```
package devonfw.tutorial.general.common.api.constants;  
  
/**  
 * Contains constants for the keys of all  
 * {@link io.oasp.module.security.common.api.accesscontrol.AccessControlPermission}s.  
 */  
public abstract class PermissionConstants {  
  
    /** {@link io.oasp.module.security.common.api.accesscontrol.AccessControlPermission} to retrieve  
     * table. */  
    public static final String FIND_TABLE = "FindTable";  
  
    /** {@link io.oasp.module.security.common.api.accesscontrol.AccessControlPermission} to save table. */  
    public static final String SAVE_TABLE = "SaveTable";  
  
    /** {@link io.oasp.module.security.common.api.accesscontrol.AccessControlPermission} to remove table.  
     */  
    public static final String DELETE_TABLE = "DeleteTable";  
}
```

[[getting-started-crud-operations\_step-5: create rest endpoints]] ===== Step 5: Create REST endpoints

Web applications need to get data from the server, so we have to expose the methods defined in the logic layer to these applications. We need a class that exposes methods as URLs to allow the applications to get the data. By convention, we call this class *<Entity>managementRestServiceImpl*.

This is an example of a REST API for our Table use case using JAX-RS.

Also note that the implementation does not follow the dogmatic *RESTFUL* approach as Devonfw proposes a more pragmatic way to use REST. Please refer to the guide [Creating Rest Service](#) for more information on the subject.

## TablemanagementRestServiceImpl.java.

```
package devonfw.tutorial.tablemanagement.service.impl.rest;

import java.util.List;

import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.BadRequestException;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.transaction.annotation.Transactional;

import devonfw.tutorial.tablemanagement.logic.api.Tablemanagement;
import devonfw.tutorial.tablemanagement.logic.api.to.TableEto;

/**
 *
 * The service class for REST calls in order to execute the methods in {@link Tablemanagement}.
 */

@Path("/tablemanagement/v1") ❶
@Named("TablemanagementRestService")
@Consumes(MediaType.APPLICATION_JSON) ❷
@Produces(MediaType.APPLICATION_JSON)
@Transactional
public class TablemanagementRestServiceImpl {

    private Tablemanagement tableManagement;

    /**
     *
     * This method sets the field <tt>tableManagement</tt>.
     *
     *
     *
     * @param tableManagement the new value of the field tableManagement
     */
    @Inject
    public void setTableManagement(Tablemanagement tableManagement) {

        this.tableManagement = tableManagement;
    }

    /**
     *
     * Delegates to {@link Tablemanagement#findTable}.
     *
     *
     *
     * @param id the ID of the {@link TableEto}
     *
     * @return the {@link TableEto}
     */
    @GET
    @Path("/table/{id}/")
    public TableEto getTable(@PathParam("id") String id) {

        Long idAsLong;

        if (id == null) {
```

```
        throw new BadRequestException("missing id");

    }

    try {
        idAsLong = Long.parseLong(id);

    } catch (NumberFormatException e) {

        throw new BadRequestException("id is not a number");

    } catch (NotFoundException e) {

        throw new BadRequestException("table not found");

    }

    return this.tableManagement.findTable(idAsLong);

}

/**
 *
 * Delegates to {@link Tablemanagement#findAllTables}.
 *
 *
 *
 * @return list of all existing restaurant {@link TableEto}s
 */

@GET
@Path("/table/")
public List<TableEto> getAllTables() {

    List<TableEto> allTables = this.tableManagement.findAllTables();

    return allTables;

}

/**
 *
 * Delegates to {@link Tablemanagement#findFreeTables}.
 *
 *
 *
 * @return list of all existing free {@link TableEto}s
 */

@GET
@Path("/freetables/")
public List<TableEto> getFreeTables() {

    return this.tableManagement.findFreeTables();

}

/**
 *
 * Delegates to {@link Tablemanagement#saveTable}.
 *
 *
 *
 * @param table the {@link TableEto} to be created
 *
 * @return the recently created {@link TableEto}
 */

@POST
```

```
@Path("/table/")
public TableEto saveTable(TableEto table) {
    return this.tableManagement.saveTable(table);
}

/**
 *
 * Delegates to {@link Tablemanagement#deleteTable}.
 *
 *
 */
* @param id ID of the {@link TableEto} to be deleted
*/
@DELETE
@Path("/table/{id}/")
public void deleteTable(@PathParam("id") Long id) {
    this.tableManagement.deleteTable(id);
}
```

- ② We send and receive the information in JSON format.
- ① We specify the version of the entire API inside its path.

As you can see, we have defined the REST URLs for our Table user case. Now, for example, you can find all tables on this URL:

```
http://<server>:<port>/application-name/services/rest/tablemanagement/v1/table/
```

### 2.17.3 DTO conversion

In the logic API, the methods of the classes should return Data Transfer Object (DTO) instead of entities. So, in OASP4J we have a mechanism to convert the entities into DTOs.

This is an example of how to convert an entity into a DTO:

```
// Conversion for lists
getBeanMapper().mapList(tableList, TableDto.class);

// Conversion for objects
getBeanMapper().map(table, TableDto.class);
```

In the example, we use the function `getBeanMapper()`. This function provides us an API to convert entities into DTOs. In the logic layer, we only have to extend the class `AbstractComponentFacade` to get access to this functionality.

[[getting-started-crud-operations\_step-6: add pagination]] ===== Step 6: Add pagination

To add pagination support to our Table CRUD, the first step is creating a new Table TO that extends the `SearchCriteriaTo` class. This class forms the foundation for every request which needs search or pagination functionality.

#### TableSearchCriteriaTo.java.

```
package devonfw.tutorial.tablemanagement.logic.api.to;

import io.oasp.module.jpa.common.api.to.SearchCriteriaTo;
```

```
import devonfw.tutorial.tablemanagement.common.api.datatype.TableState;

/**
 *
 * This is the {@link SearchCriteriaTo} search criteria} {@link
net.sf.mmm.util.transferobject.api.TransferObject TO}
 */

public class TableSearchCriteriaTo extends SearchCriteriaTo {

    /** UID for serialization. */
    private static final long serialVersionUID = 1L;

    private Long waiterId;

    private Long number;

    private TableState state;

    /**
     *
     * The constructor.
     */
    public TableSearchCriteriaTo() {
        super();
    }

    /**
     *
     * @return waiterId
     */
    public Long getWaiterId() {
        return this.waiterId;
    }

    /**
     *
     * @param waiterId the waiterId to set
     */
    public void setWaiterId(Long waiterId) {
        this.waiterId = waiterId;
    }

    /**
     *
     * @return state
     */
    public TableState getState() {
        return this.state;
    }

    /**
     *
     * @param state the state to set
     */
    public void setState(TableState state) {
```

```
        this.state = state;
    }

    /**
     *
     * @return number
     */

    public Long getNumber() {
        return this.number;
    }

    /**
     *
     * @param number the number to set
     */

    public void setNumber(Long number) {
        this.number = number;
    }
}
```

Now we will create a new POST REST endpoint (pagination request have to be POST) in our TablemanagementRestServiceImpl class.

```
/**
 * Delegates to {@link Tablemanagement#findTableEtos}.
 *
 * @param searchCriteriaTo the pagination and search criteria to be used for finding tables.
 * @return the {@link PaginatedListTo list} of matching {@link TableEto}s.
 */
@Path("/table/search")
@POST
public PaginatedListTo<TableEto> findTablesByPost(TableSearchCriteriaTo searchCriteriaTo) {
    return this.tableManagement.findTableEtos(searchCriteriaTo);
}
```

### Note

Make sure to press **Ctrl-Shift-O** after inserting this new method, to make Eclipse auto-import the dependencies of `PaginatedListTo` and `TableSearchCriteriaTo`.

Consequently we have to declare this new method `findTableEtos` in the table management classes in our logic layer:

### Tablemanagement.java.

```
/**
 * Returns a list of restaurant tables matching the search criteria.
 *
 * @param criteria the {@link TableSearchCriteriaTo}.
 * @return the {@link List} of matching {@link TableEto}s.
 */
PaginatedListTo<TableEto> findTableEtos(TableSearchCriteriaTo criteria);
```

### TablemanagementImpl.java.

```
@Override
```

```
public PaginatedListTo<TableEto> findTableEtos(TableSearchCriteriaTo criteria) {
    criteria.limitMaximumPageSize(MAXIMUM_HIT_LIMIT); ❶
    PaginatedListTo<TableEntity> tables = getTableDao().findTables(criteria);

    return mapPaginatedEntityList(tables, TableEto.class);
}
```

- ❶ As you can see, we have limited the maximum results per page to prevent clients from requesting pages with too big a size.

And finally, we have to define our pagination method in our DAO class.

### TableDao.java.

```
/**
 * Finds the {@link TableEntity orders} matching the given {@link TableSearchCriteriaTo}.
 *
 * @param criteria is the {@link TableSearchCriteriaTo}.
 * @return the {@link List} with the matching {@link TableEntity} objects.
 */
PaginatedListTo<TableEntity> findTables(TableSearchCriteriaTo criteria);
```

### TableDaoImpl.java.

```
@Override
public PaginatedListTo<TableEntity> findTables(TableSearchCriteriaTo criteria) {

    TableEntity table = Alias.alias(TableEntity.class);
    EntityPathBase<TableEntity> alias = Alias.$(table);
    JPAQuery query = new JPAQuery(getEntityManager()).from(alias);

    Long waiterId = criteria.getWaiterId();
    if (waiterId != null) {
        query.where(Alias.$(table.getWaiterId()).eq(waiterId));
    }
    Long number = criteria.getNumber();
    if (number != null) {
        query.where(Alias.$(table.getNumber()).eq(number));
    }
    TableState state = criteria.getState();
    if (state != null) {
        query.where(Alias.$(table.getState()).eq(state));
    }

    return findPaginated(criteria, query, alias);
}
```

### Note

While auto-completing the new imports using **Ctrl-Shift-O** after adding the above methods, select `com.mysema.query.alias` as the import for the `Alias` class.

In this case, we have used QueryDSL to create the query. You can read more about QueryDSL at [www.querydsl.com](http://www.querydsl.com).

[[getting-started-crud-operations\_step-7: sort the results]] ===== Step 7: Sort the results

In OASp4J exists a special TO (Transfer Object) called 'OrderByTo' to transmit sorting parameters from client to server. This is the JSON format that the server expects when using this TO:

```
{
    sort: [
        {
            name: "sortingCriterial",
```

```

        direction:"ASC"
    },
{
    name:"sortingCriteria2",
    direction:"DESC"
},
...
]
}

```

Devonfw proposes to use POST as the HTTP method for endpoints implementing search or pagination support.

By default, in Devonfw, `SearchCriteriaTo` class is already embedding this sorting TO, so we only need to manage sorting in `TableDaoImpl.java` because our pagination method does not need any modification.

If our method needs sorting but not pagination we need to manually add to our own transfer object the following variable (and its setter and getter methods):

```
private List<OrderByTo> sort;
```

We are going to modify the method `findTables` in our `TableDaoImpl`. Insert the following line right before the final `return` statement:

```
// Add order by fields
addOrderBy(query, alias, table, criteria.getSort());
```

Now add the following method to `TableDaoImpl`:

```

private void addOrderBy(JPAQuery query, EntityPathBase<TableEntity> alias, TableEntity table,
List<OrderByTo> sort) {

    if (sort != null && !sort.isEmpty()) {
        for (OrderByTo orderEntry : sort) {
            if ("number".equals(orderEntry.getName())) {

                if (OrderDirection.ASC.equals(orderEntry.getDirection())) {
                    query.orderBy(Alias.$(table.getNumber()).asc());
                } else {
                    query.orderBy(Alias.$(table.getNumber()).desc());
                }

            } else if ("waiterId".equals(orderEntry.getName())) {

                if (OrderDirection.ASC.equals(orderEntry.getDirection())) {
                    query.orderBy(Alias.$(table.getWaiterId()).asc());
                } else {
                    query.orderBy(Alias.$(table.getWaiterId()).desc());
                }

            } else if ("state".equals(orderEntry.getName())) {

                if (OrderDirection.ASC.equals(orderEntry.getDirection())) {
                    query.orderBy(Alias.$(table.getState()).asc());
                } else {
                    query.orderBy(Alias.$(table.getState()).desc());
                }
            }
        }
    }
}

```

As you can see, we have added a private method to add sorting filter to our query depending on the sort parameters received.

## [[getting-started-crud-operations\_step-8: test the example]] ===== Step 8: Test the example

In order to test the example we are going to use the user `chief` to obtain the tables. To be able to access to that data we need first to grant permissions to the `chief` user. We can do it specifying the role and the permissions in the `access-control-schema.xml` file located in `src/main/resources/config/app/security/`.

```
<group id="Chief" type="role">
  <permissions>
    <permission id="FindTable"/>
  </permissions>
</group>
```

Now if we run the application we can access to the tables data with the URL

```
http://<server>/<app>/services/rest/tablemanagement/v1/table/
```

And, after logging as `chief`, the server response should be:

```
[{"id":101,"modificationCounter":1,"revision":null,"waiterId":null,"number":1,"state":"OCCUPIED"}, {"id":102,"modificationCounter":1,"revision":null,"waiterId":null,"number":2,"state":"FREE"}, {"id":103,"modificationCounter":1,"revision":null,"waiterId":null,"number":3,"state":"FREE"}, {"id":104,"modificationCounter":1,"revision":null,"waiterId":null,"number":4,"state":"FREE"}, {"id":105,"modificationCounter":1,"revision":null,"waiterId":null,"number":5,"state":"FREE"}]
```

## 2.18 Getting Started Cobigen

In Devonfw we have a server-side code generator called Cobigen. Cobigen is capable to create CRUD code from an entity or generate the content of the class that defines the user permissions. Cobigen is distributed in the Devonfw distribution as an Eclipse plugin, and is available to all Devonfw developers.

If you want to go deeper in Cobigen you can visit the documentation of the [Cobigen core](#).

[[getting-started-cobigen\_preparing-cobigen for first use]] == Preparing Cobigen for first use

Before you can use Cobigen, you have to install the templates to be used by Cobigen. The Devonfw distribution comes with a set of default templates in the directory `workspaces\main\CoBiGen_Templates`.

1. Open Eclipse by executing the batch file `eclipse-main.bat`
2. Select "File - Import"
3. Select "General - Existing projects into workspace"
4. Select the directory `workspaces\main\CoBiGen_Templates`
5. Finish the import.

### Note

In your own projects, you can create additional templates. Please refer to the document [CoBiGen.pdf](#) for further information.

[[getting-started-cobigen\_creating-a crud with cobigen]] == Creating a CRUD with Cobigen

In an earlier chapter about [CRUD functionality](#) you saw the individual steps necessary to implement a basic CRUD functionality.

Using Cobigen, you can save most of these steps, and get a working result in far less time. We are going to explain how to use Cobigen to generate the code and classes related to the CRUD operations of an entity but you can know more about the [Cobigen usage in Eclipse](#).

Cobigen needs a starting point to generate the code of a CRUD case. In this example the starting point is the `StaffMemberEntity` class, modelling a member of the staff of our restaurant. So we are going to create a CRUD for the new `StaffMemberEntity` class.

[[getting-started-cobigen\_step-1: entity creation]] ===== Step 1: Entity creation

We are going to create a `StaffMember` entity. First, we are going to add the database schema to our database.

In the script `resources/db/migration/V0001__Create_schema.sql` we add:

```
CREATE TABLE STAFFMEMBER(
    id BIGINT NOT NULL,
    modificationCounter INTEGER NOT NULL,
    firstname VARCHAR(255),
    lastname VARCHAR(255),
    login VARCHAR(255)
```

```
);
```

And in the same path, we are going to create a new file (if it do not exist) to add the default data to the StaffMember created. We create `V0002_Master_data.sql` file.

```
INSERT INTO STAFFMEMBER (id, login, firstname, lastname, modificationCounter) VALUES
(0, 'chief', 'Charly', 'Chief', 0);
INSERT INTO STAFFMEMBER (id, login, firstname, lastname, modificationCounter) VALUES
(1, 'cook', 'Carl', 'Cook', 0);
INSERT INTO STAFFMEMBER (id, login, firstname, lastname, modificationCounter) VALUES
(2, 'waiter', 'Willy', 'Waiter', 0);
INSERT INTO STAFFMEMBER (id, login, firstname, lastname, modificationCounter) VALUES
(3, 'barkeeper', 'Bianca', 'Barkeeper', 0);
```

Now, we create a new StaffMember entity in the package `devonfw.tutorial.staffmanagement.dataaccess.api` with the following code:

### StaffMemberEntity.java.

```
package devonfw.tutorial.staffmanagement.dataaccess.api;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

import devonfw.tutorial.general.dataaccess.api.ApplicationPersistenceEntity;
import devonfw.tutorial.general.dataaccess.api.StaffMember;

//Add imports with respect to the package structure.

/**
 *
 * The {@link devonfw.tutorial.general.dataaccess.api.ApplicationPersistenceEntity} persistent entity
 * for
 *
 * {@link StaffMember}.
 */

@Entity
@Table(name = "StaffMember")
public class StaffMemberEntity extends ApplicationPersistenceEntity implements StaffMember {

    private static final long serialVersionUID = 1L;

    private String name;

    private String firstName;

    private String lastName;

    /**
     * The constructor.
     */
    public StaffMemberEntity() {

        super();
    }

    @Column(name = "login", unique = true)
    @Override
    public String getName() {

        return this.name;
    }

    @Override
    public void setName(String login) {
```

```
    this.name = login;
}

@Override
public String getFirstName() {

    return this.firstName;
}

@Override
public void setFirstName(String firstName) {

    this.firstName = firstName;
}

@Override
public String getLastName() {

    return this.lastName;
}

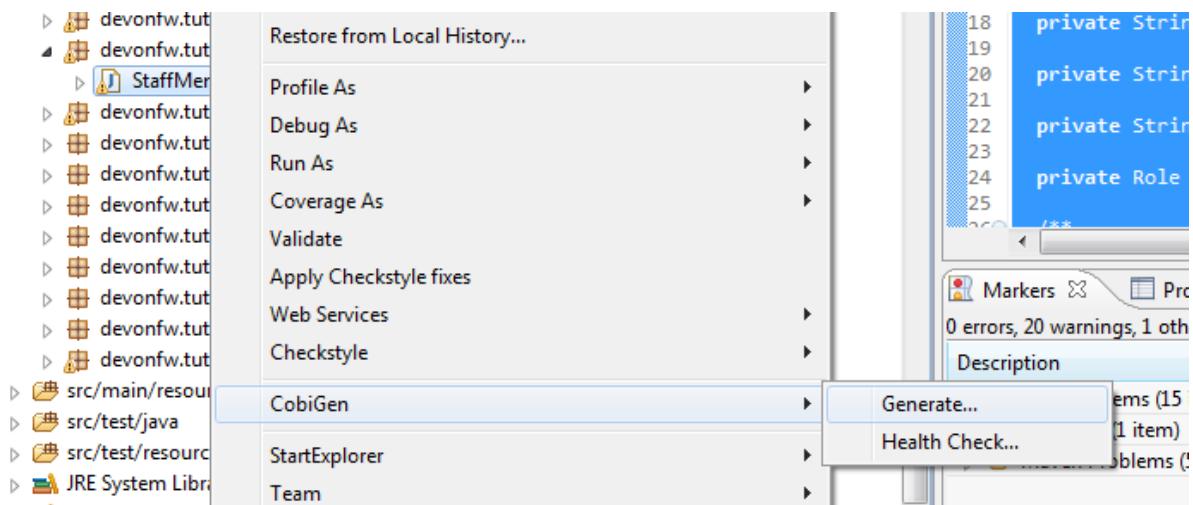
@Override
public void setLastName(String lastName) {

    this.lastName = lastName;
}

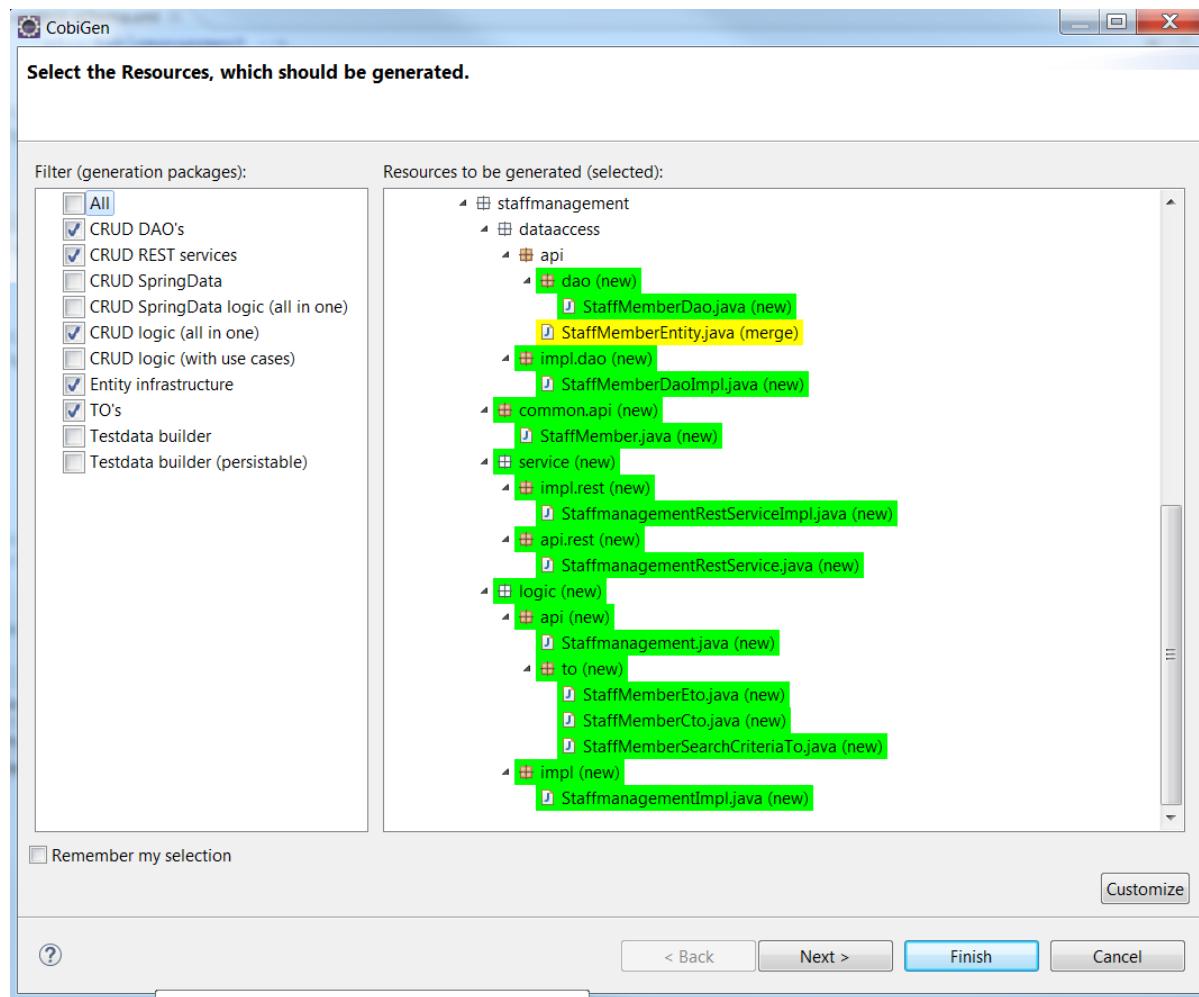
}
```

[[getting-started-cobigen\_step-2: generate classes]] ===== Step 2: Generate classes

To generate the rest of the classes concerning the StaffMember CRUD, we only have to do a right click on the StaffMemberEntity.java class in Eclipse Project Explorer and select "CobiGen ' Generate".



This action opens a code generator wizard, like this:

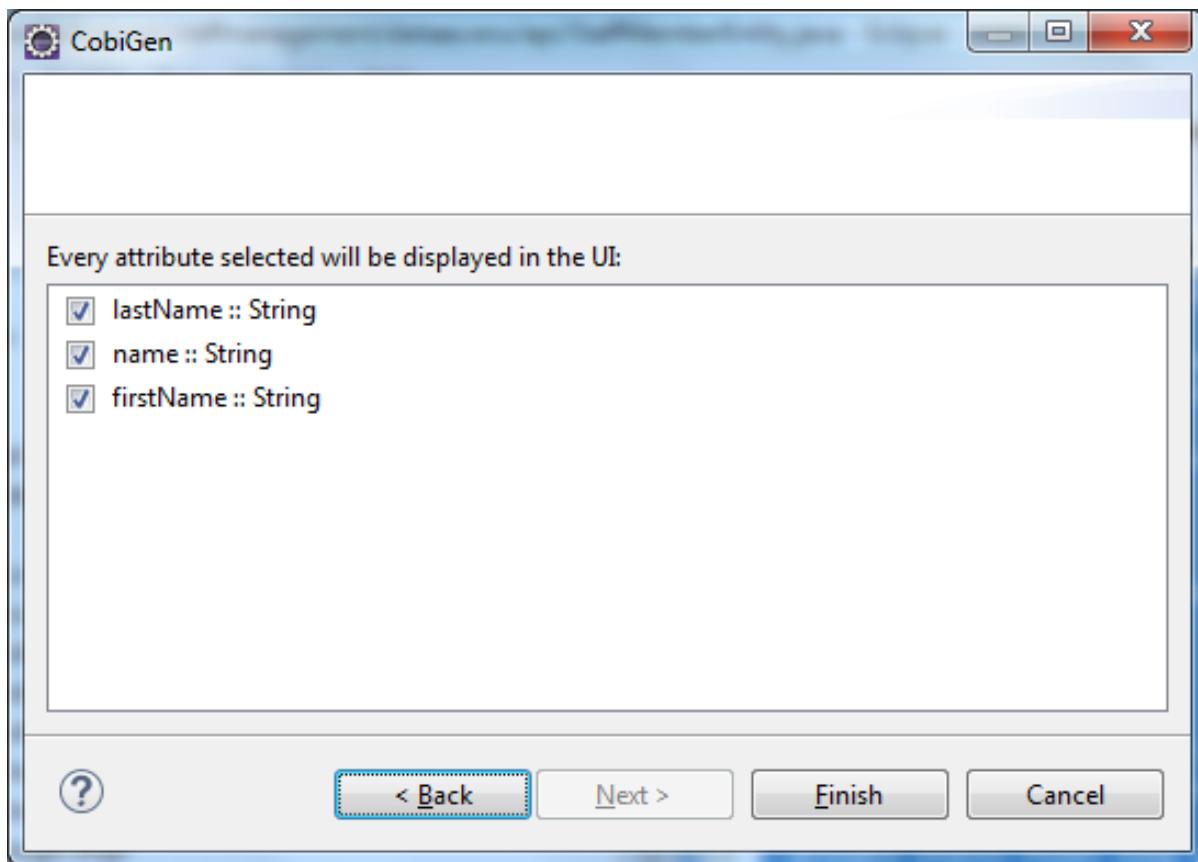


In this wizard you can select which classes you want to generate, organized by layer. In this example, please select:

- CRUD DAO's
- CRUD REST services
- CRUD logic layer (all in one)
- Entity infrastructure
- TO's

and continue.

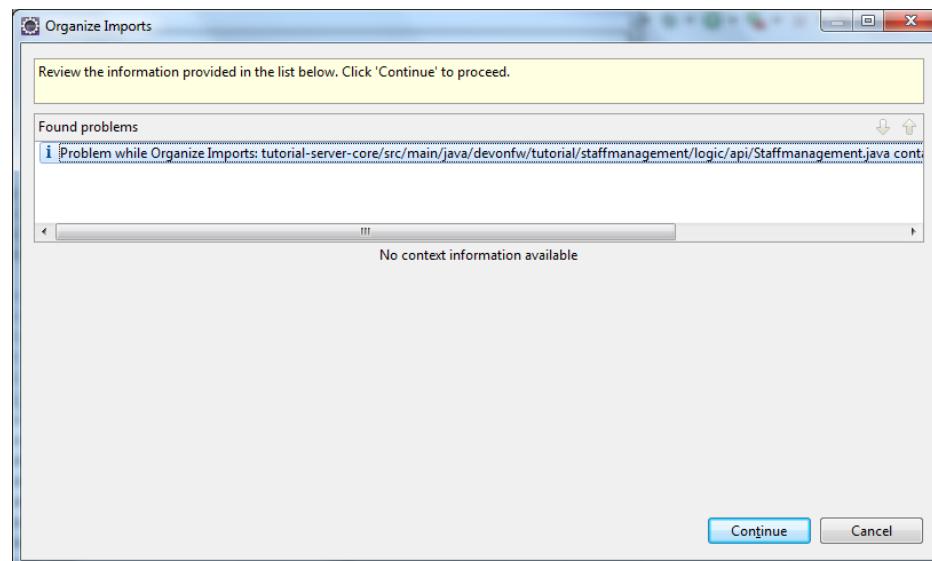
In the next step you can select the fields of the entity that you want to expose via the REST service.



Afterwards, click on "Finish" to let CobiGen do its work.

### Note

It is possible that you will see a final dialog containing some warnings about ambiguous imports. You should review the mentioned files, and fix the imports yourself.



In many cases, the imports are easily fixable by letting Eclipse auto-complete them by pressing "Ctrl-Shift-O".

Cobigen also works incrementally. Cobigen merges your changes and updates all classes based on the Entity class' fields. So you can use Cobigen to generate the structure and the different classes and then develop custom parts of your CRUD.

[[getting-started-cobigen\_creating-user permissions]] == Creating user permissions

In OASP4J applications the roles and permissions are defined by the *PermissionConstants* class. The content of this class is bound with the permissions defined in the *access-control-schema.xml* file. Cobigen let us to automatically generate (or update) the content of the *PermissionConstants* class from the *access-control-schema.xml* content. To achieve this we only have to follow two simple steps.

[[getting-started-cobigen\_step-1: define the permissions and roles]] ===== Step 1: Define the permissions and roles

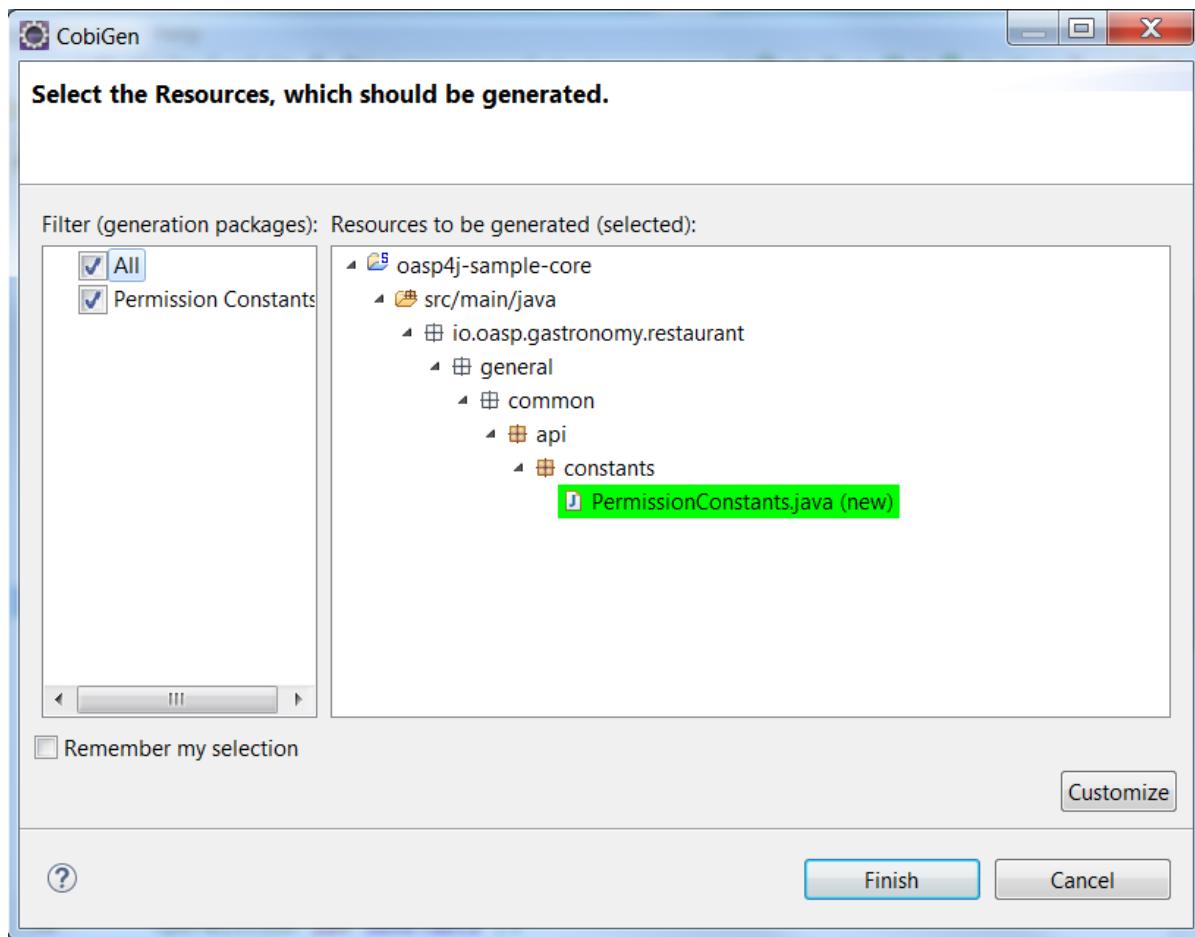
In Eclipse open the *access-control-schema.xml* located in */oasp4j-sample-core/src/main/resources/config/app/security/access-control-schema.xml* and define the permissions to the roles or group of roles like:

```
<group id="MasterData" type="group">
  <permissions>
    <!-- staffmembermanagement -->
    <permission id="FindStaffMember"/>
    <permission id="SaveStaffMember"/>
    <permission id="DeleteStaffMember"/>
  </permissions>
</group>
```

[[getting-started-cobigen\_step-2: generate the permissionconstants class]] ===== Step 2: Generate the *PermissionConstants* class

Right click on the *access-control-schema.xml* and select *Cobigen > Generate...*

This action opens a code generator wizard, like this:

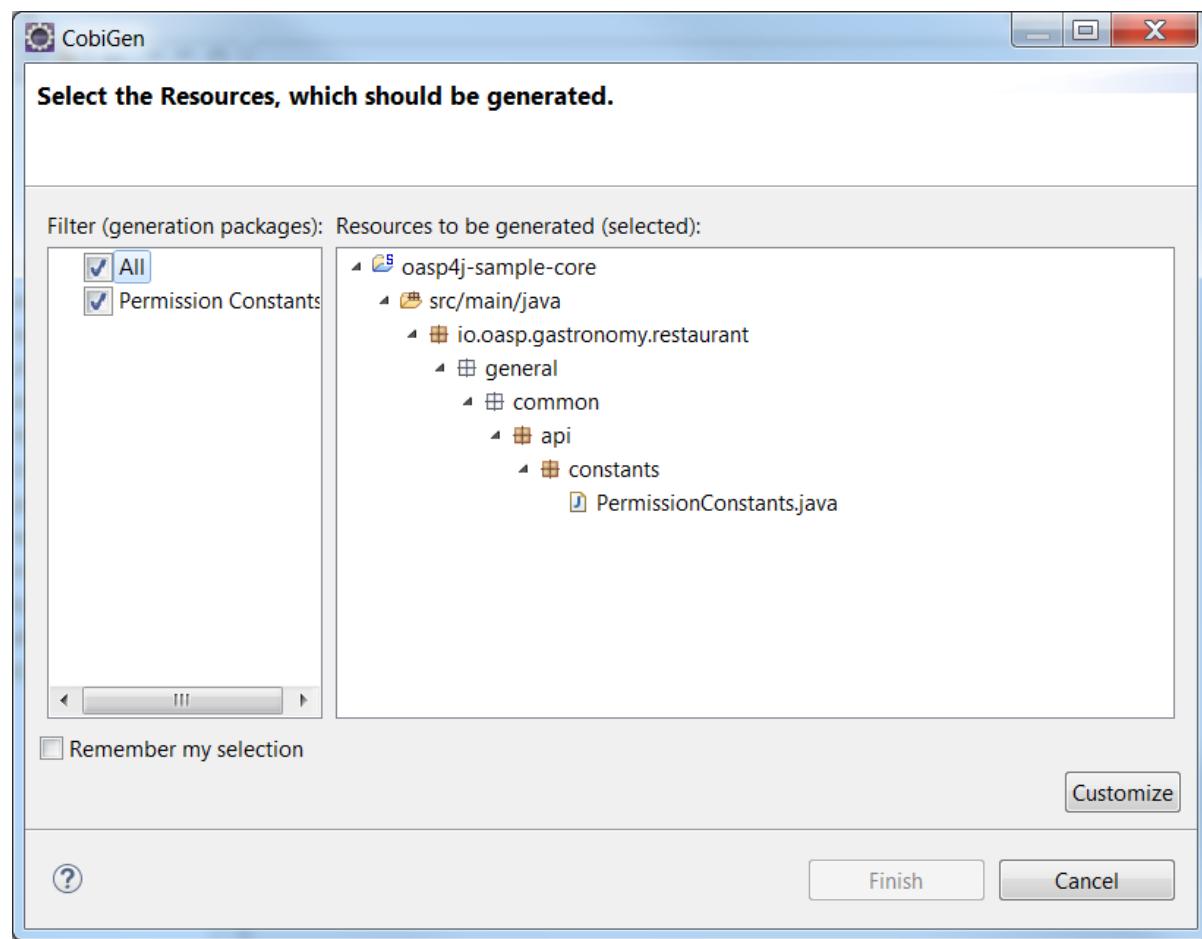


In this case you have only one option. Select *Permissions Constants* and press *Finish*. You should see now the new Permissions added in the file */oasp4j-sample-core/src/main/java/io/oasp/gastronomy/restaurant/general/common/api/constants/PermissionConstants.java*

```
public static final String FIND_STAFFMEMBER = "FindStaffMember";
public static final String SAVE_STAFFMEMBER = "SaveStaffMember";
public static final String DELETE_STAFFMEMBER = "DeleteStaffMember";
```

#### Note

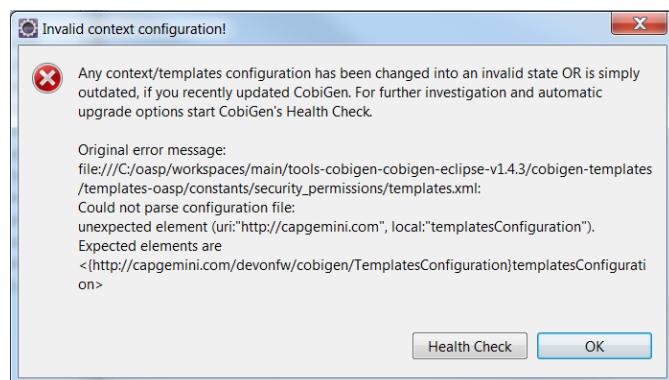
It is possible that you can't press *Finish* button in CobiGen.



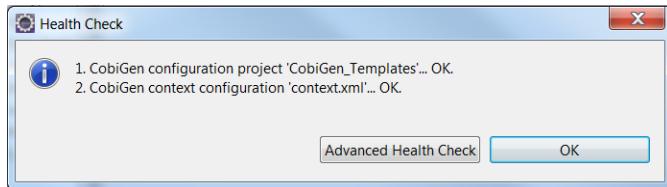
This happens because you are using an old version of CobiGen and the wizard can't merge the class *PermissionConstants*. To work around this you need to delete the class *PermissionConstants.java* and try again. Cobigen will generate for us the class and will fill it with the updated content.

#### [[getting-started-cobigen\_fixing-context problems]] === Fixing context problems

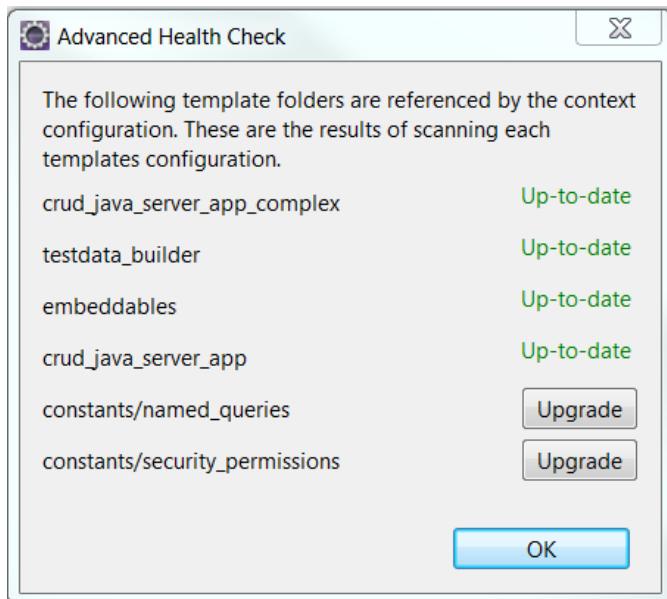
When launching the *Cobigen > Generate* wizard you may find problems related to the context, like the following one



This happens because you need to update the templates. So do again right click on the *access-control-schema.xml* and select this time the *Cobigen > Health Check* option and you will see a window with a message like the following



Click in *Advance Health Check*



Now upgrade the template to *constants/security\_permissions* and press *OK*. You now should be able to use Cobigen to generate the *PermissionConstants* class content.

## 2.19 Transfer-Objects

The technical data model is defined in form of [persistent entities](#). However, passing persistent entities via *call-by-reference* across the entire application will soon cause problems:

- Changes to a persistent entity are directly written back to the persistent store when the transaction is committed. When the entity is send across the application also changes tend to take place in multiple places endangering data sovereignty and leading to inconsistency.
- You want to send and receive data via services across the network and have to define what section of your data is actually transferred. If you have relations in your technical model you quickly end up loading and transferring way too much data.
- Modifications to your technical data model shall not automatically have impact on your external services causing incompatibilities.

To prevent such problems transfer-objects are used leading to a *call-by-value* model and decoupling changes to persistent entities.

### 2.19.1 Business-Transfer-Objects

For each [persistent entity](#) we create or generate a corresponding *entity transfer object* (ETO) that has the same properties except for relations. In order to centralize the properties (getters and setters with their javadoc) we use a common interface for the entity and its ETO.

If we need to pass an entity with its relation(s) we create a corresponding *composite transfer object* (CTO) that only contains other transfer-objects or collections of them. This pattern is illustrated by the following UML diagram from our sample application.

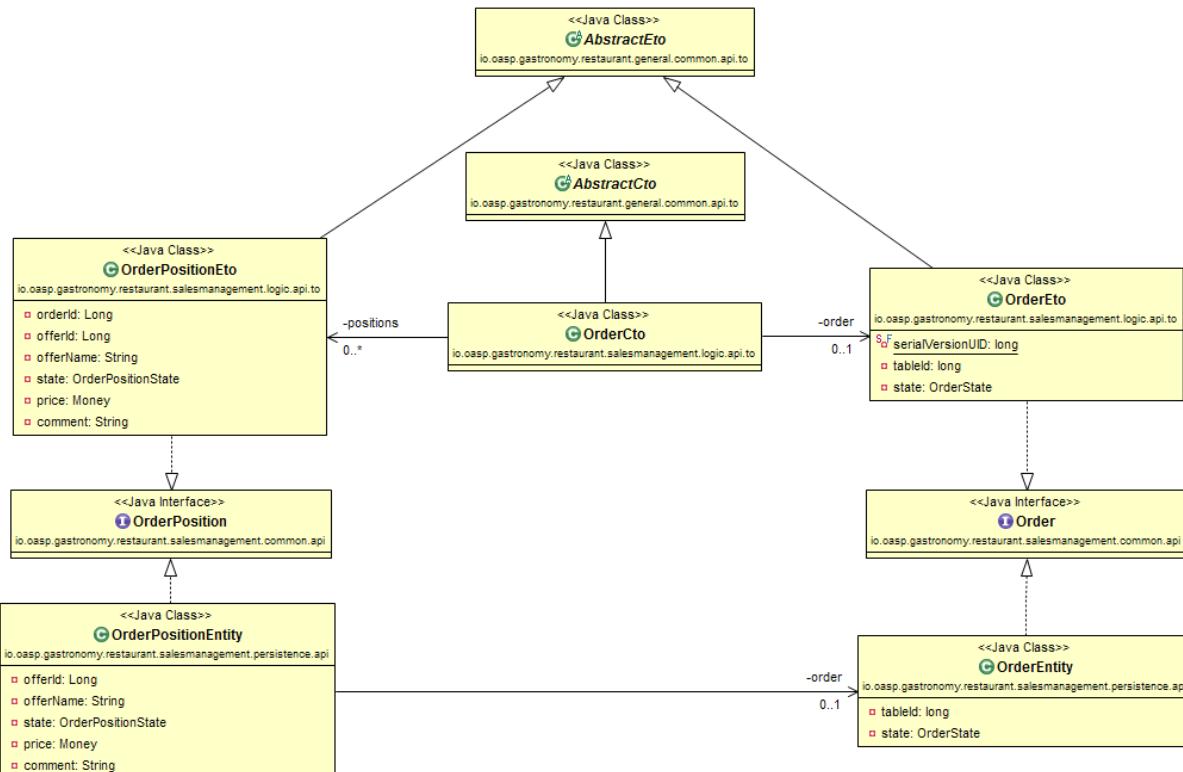


Figure 2.1. ETOs and CTOs

Finally, there are typically transfer-objects for data that is never persistent. A common example are search criteria objects (derived from `SearchCriteriaTo` in our sample application).

The [logic layer](#) defines these transfer-objects (ETOs, CTOs, etc.) and will only pass such objects instead of [persistent entities](#).

## 2.19.2 Service-Transfer-Objects

If we need to do [service versioning](#) and support previous APIs or for external services with a different view on the data, we create separate transfer-objects to keep the service API stable (see [service layer](#)).

## 2.20 Logic Layer

The logic layer is the heart of the application and contains the main business logic. According to the OAPS4J [business architecture](#) we should divide an application into *business components*. The *component part* assigned to the logic layer contains the functional use-cases the business component is responsible for. For further understanding consult the [application architecture](#).

### 2.20.1 Component Interface

A component may consist of several use cases but is only accessed by the next higher layer or other components through one interface, i.e. by using one *Spring bean*.

If the implementation of the component interface gets too complex it is recommended to further subdivide it in separate use-case-interfaces to be aggregated in the main component interface. This suits for better maintainability.

First we create an interface that contains the method(s) with the business operation documented with JavaDoc. The API of the use cases has to be business oriented. This means that all parameters and return types of a use case method have to be business [transfer-objects](#), [datatypes](#) (*String*, *Integer*, *MyCustomerNumber*, etc.), or collections of these. The API may only access objects from other business components in the (transitive) [dependencies](#) of the declaring business component. Here is an example of a use case interface:

```
public interface StaffManagement {  
  
    StaffMemberEto getStaffMemberByLogin(String login);  
  
    StaffMemberEto getStaffMember(Long id);  
  
    ...  
}
```

### 2.20.2 Component Implementation

The implementation of the use case typically needs access to the persistent data. This is done by [injecting](#) the corresponding [DAO](#). For the [principle data sovereignty](#) only DAOs of the same business component may be accessed directly from the use case. For accessing data from other components the use case has to use the corresponding [component interface](#). Further it shall not expose persistent entities from the persistence layer and has to map them to [transfer objects](#).

Within a use-case implementation, entities are mapped via a *BeanMapper* to [persistent entities](#). Let's take a quick look at some of the StaffManagement methods:

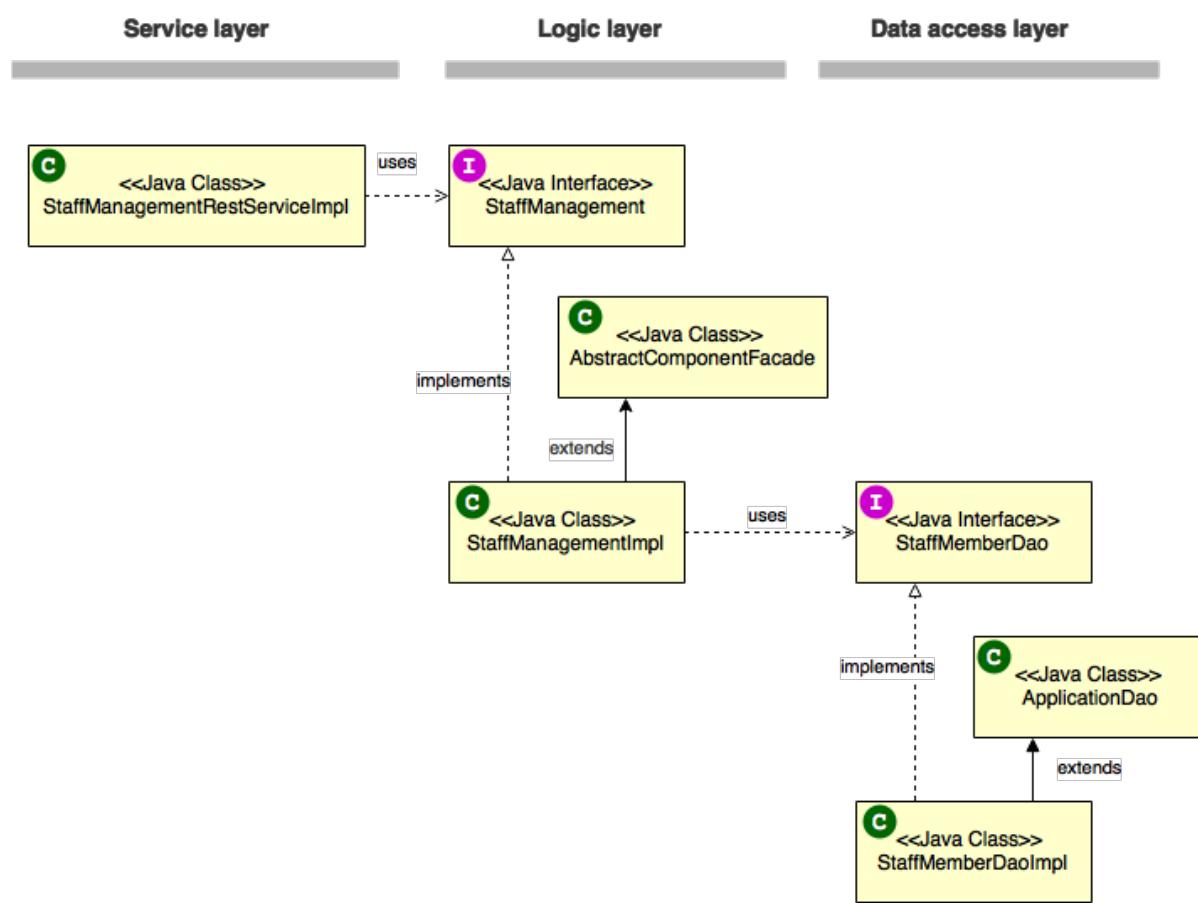
```
package io.oasp.gastronomy.restaurant.staffmanagement.logic.impl;  
  
public class StaffManagementImpl extends AbstractComponentFacade implements StaffManagement {  
  
    public StaffMemberEto getStaffMemberByLogin(String login) {  
        StaffMemberEntity staffMember = getStaffMemberDao().searchByLogin(login);  
        return getBeanMapper().map(staffMember, StaffMemberEto.class);  
    }  
  
    public StaffMemberEto getStaffMember(Long id) {  
        StaffMemberEntity staffMember = getStaffMemberDao().find(id);  
        return getBeanMapper().map(staffMember, StaffMemberEto.class);  
    }  
}
```

As you can see, provided entities are mapped to corresponding business objects (here `StaffMemberEto.class`). These business objects are simple POJOs (Plain Old Java Objects) and stored in: `<package-name-prefix>.<domain>.<application-name>.<component>.api..`

The mapping process of these entities and the declaration of the `AbstractLayerImpl` class are described [here](#). For every business object there has to be a mapping entry in the `src/main/resources/config/app/common/dozer-mapping.xml` file. For example, the mapping entry of a `TableEto` to a `Table` looks like this:

```
<mapping>
  <class-a>io.oasp.gastronomy.restaurant.tablemanagement.logic.api.TableEto</class-a>
  <class-b>io.oasp.gastronomy.restaurant.tablemanagement.persistence.api.entity.Table</class-b>
</mapping>
```

Below, a class diagram illustrating the pattern is shown (here: the `StaffManagement` business component):



As the picture above illustrates, the necessary [DAO](#) entity to access the database is provided by an abstract class. Use Cases that need access to this DAO entity, have to extend that abstract class. Needed dependencies (in this case the `staffMemberDao`) are resolved by Spring, see [here](#). For the validation (e.g. to check if all needed attributes of the `StaffMember` have been set) either Java code or [Drools](#), a business rule management system, can be used.

### 2.20.3 Passing Parameters Among Components

[Entities](#) have to be detached for the reasons of data sovereignty, if entities are passed among components or [layers](#) (to service layer). For further details see [Bean-Mapping](#). Therefore we are using [transfer-objects](#) (TO) with the same attributes as the entity that is persisted. The packages are:

Persistence Entities	<package-name-prefix>.<domain>.<application-name>.<component>.persistence.api.entity
Transfer Objects(TOs)	<package-name-prefix>.<domain>.<application-name>.<component>.logic.api

This mapping is a simple copy process. So changes out of the scope of the owning component to any TO do not directly affect the persistent entity.

## 2.20.4 Security

The logic layer is the heart of the application. It is also responsible for authorization and hence security is important here.

### Direct Object References

A security threat are [Insecure Direct Object References](#). This simply gives you two options:

- avoid direct object references at all
- ensure that direct object references are secure

Especially when using REST, direct object references via technical IDs are common sense. This implies that you have a proper [authorization](#) in place. This is especially tricky when your authorization does not only rely on the type of the data and according static permissions but also on the data itself. Vulnerabilities for this threat can easily happen by design flaws and inadvertence. Here an example from our sample application:

### TablemanagementImpl.java.

```

@RolesAllowed(PermissionConstants.FIND_TABLE)
public TableEto findTable(Long id) {

    return getBeanMapper().map(getTableDao().findOne(id), TableEto.class);
}

```

We have a generic use-case to manage *Tables*. In the first place it makes sense to write a generic REST service to load and save these *Tables*. However, the permission to read or even update such *Table* depend on the business object hosting the Table. Therefore such a generic REST service would open the door for this OWASP A4 vulnerability. To solve this in a secure way you need individual services for each hosting business object. There you have to check permissions based on the parent business object. In this example the ID of the Table would be the direct object reference and the ID of the business object would be the indirect object reference.

## 2.21 Service Layer

The service layer is responsible to expose functionality of the [logical layer](#) to external consumers over a network via [technical protocols](#).

[[getting-started-creating-rest-service\_types-of services]] == Types of Services

If you want to create a service please distinguish the following types of services:

- **External Services**

are used for communication between different companies, vendors, or partners.

- **Internal Services**

are used for communication between different applications in the same application landscape of the same vendor.

- **Back-end Services**

are internal services between Java back-ends typically with different release and deployment cycles (otherwise if not Java consider this as external service).

- **JS-Client Services**

are internal services provided by the Java back-end for JavaScript clients (GUI).

- **Java-Client Services**

are internal services provided by the Java back-end for a native Java client (JavaFx, EclipseRcp, etc.).

The choices for technology and protocols will depend on the type of service. Therefore the following table gives a guideline for aspects according to the service types. These aspects are described below.

Table 2.3. Aspects according to service-type

Aspect	External Service	Back-end Service	JS-Client Service	Java-Client Service
<a href="#">Versioning</a>	required	required	not required	not required
<a href="#">Interoperability</a>	mandatory	not required	implicit	not required
<a href="#">recommended Protocol</a>	<a href="#">SOAP or REST</a>	<a href="#">REST</a>	<a href="#">REST+JSON</a>	<a href="#">REST</a>

### 2.21.1 Versioning

For services consumed by other applications we use versioning to prevent incompatibilities between applications when deploying updates. This is done by the following conventions:

- We define a version number and prefix with v for version (e.g. v1).
- If we support previous versions we use that version numbers as part of the Java package defining the service API (e.g. com.foo.application.component.service.api.v1)
- We use the version number as part of the service name in the remote URL (e.g. <https://application.foo.com/services/rest/component/v1/resource>)
- Whenever we need to change the API of a service in an incompatible we increment the version (e.g. v2) as an isolated copy of the previous version of the service. In the implementation of different

versions of the same service we can place compatibility code and delegate to the same unversioned use-case of the logic layer whenever possible.

- For maintenance and simplicity we avoid keeping more than one previous version.

## 2.21.2 Interoperability

For services that are consumed by clients with different technology *interoperability* is required. This is addressed by selecting the right [protocol](#) following protocol-specific best practices and following our considerations especially *simplicity*.

## 2.21.3 Protocol

For services there are different protocols. Those relevant for and recommended by Devonfw are listed in the following sections with examples how to implement them in Java.

### REST

REST is not a technology and certainly is not a standard of some kind. It is merely an architectural style that chalks down how to write a web service in a certain way. REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON, XML. JSON is the most popular one. Principal protagonist of a REST architecture is a "*Resource*" which can be uniquely identified by an Uniform Resource Identifier or URI. State of a resource at any given point of time is represented by a document and is called Representation of resource. The client can update the state of resource by transferring the representation along with the request. The new representation is now returned to client along with the response. The representation contains the information in formats like html, xml, JSON etc that is accepted by the resource. The resource which adheres to rules of REST architecture is called a RESTful resource and web service that adheres to this rule are called RESTful web service.

### SOAP

SOAP is a common protocol that is rather complex and heavy. It allows to build inter-operable and well specified services (see WSDL). SOAP is transport neutral what is not only an advantage. We strongly recommend to use HTTPS transport and ignore additional complex standards like WS-Security and use established HTTP-Standards such as RFC2617 (and RFC5280).

[[getting-started-creating-rest-service\_details-on rest service.]] == Details on Rest Service.

For a general introduction consult the [wikipedia](#). For CRUD operations in REST we distinguish between collection and element URIs:

- A collection URI is build from the rest service URI by appending the name of a collection. This is typically the name of an entity. Such URI identifies the entire collection of all elements of this type. Example:

```
https://mydomain.com/myapp/services/rest/mycomponent/v1/myentity
```

- An element URI is build from a collection URI by appending an element ID. It identifies a single element (entity) within the collection. Example:

```
https://mydomain.com/myapp/services/rest/mycomponent/v1/myentity/42
```

### [[getting-started-creating-rest-service\_representation-of resource]] === Representation of Resource

A resource in REST is similar Object in Object Oriented Programming or similar to Entity in a Database. Once a resource is identified then its representation is to be decided using a standard format so that server can send the resource in above said format and client can understand the same format. For example, in RESTful Web Services , " User " is a resource which is represented using following XML format:

```
<user>
  <id>1</id>
  <name>Mahesh</name>
  <profession>Teacher</profession>
</user>
```

Same resource can be represented in JSON format:

```
{
  "id":1,
  "name": "Mahesh",
  "profession": "Teacher"
}
```

### [[getting-started-creating-rest-service\_characteristics-of good representation]] === Characteristics of good Representation

In REST, there is no restriction on the format of a resource representation. A client can ask for JSON representation where as another client may ask for XML representation of same resource to the server and so on. It is responsibility of the REST server to pass the client the resource in the format that client understands. Following are important points to be considered while designing a representation format of a resource in a RESTful web services.

**Understandability:** Both Server and Client should be able to understand and utilize the representation format of the resource.

**Completeness:** Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.

**Linkability:** A resource can have a linkage to another resource, a format should be able to handle such situations.

### [[getting-started-creating-rest-service\_restful-web services-messages]] === RESTful Web Services-Messages

RESTful web services make use of HTTP protocol as a medium of communication between client and server. A client sends a message in form of a HTTP Request and server responds in form of a HTTP Response. This technique is terms as Messaging. These messages contain message data and metadata that is information about message itself. Let's have a look on HTTP Request and HTTP Response messages for HTTP 1.1.

A HTTP Request has five major parts:

- Verb- Indicate HTTP methods such as GET, POST etc.
- URI- Contains the URI, Uniform Resource Identifier to identify the resource on server

- HTTP Version- Indicate HTTP version, for example HTTP v1.1 .
- Request Header- Contains metadata for the HTTP Request message as key-value pairs. For example, client ( or browser) type, format supported by client, format of message body, cache settings etc.
- Request Body- Message content or Resource representation.

## HTTP RESPONSE

A HTTP Response has four major parts:

- Status/Response Code- Indicate Server status for the requested resource. For example 404 means resource not found and 200 means response is ok.
- HTTP Version- Indicate HTTP version, for example HTTP v1.1 .
- Response Header- Contains metadata for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.
- Response Body- Response message content or Resource representation.

[[getting-started-creating-rest-service\_constructing-a standard uri]] === Constructing a standard URI

Addressing refers to locating a resource or multiple resources lying on the server. It is analogous to locate a postal address of a person.

Each resource in REST architecture is identified by its URI, Uniform Resource Identifier. A URI is of following format:

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

Purpose of an URI is to locate a resource(s) on the server hosting the web service. Another important attribute of a request is VERB which identifies the operation to be performed on the resource. For example, in RESTful Web Services - First Application tutorial, URI is <http://localhost:8080/UserManagement/rest/UserService/users> and VERB is GET.

Following are important points to be considered while designing a URI:

- Use Plural Noun - Use plural noun to define resources. For example, we've used users to identify users as a resource.
- Avoid using spaces - Use underscore(\_) or hyphen(-) when using a long resource name, for example, use authorized\_users instead of authorized%20users.
- Use lowercase letters - Although URI is case-insensitive, it is good practice to keep url in lower case letters only.
- Maintain Backward Compatibility - As Web Service is a public service, a URI once made public should always be available. In case, URI gets updated, redirect the older URI to new URI using HTTP Status code, 300.
- Use HTTP Verb - Always use HTTP Verb like GET, PUT, and DELETE to do the operations on the resource. It is not good to use operations names in URI.

The "pure" REST architecture style is more suitable for creating "scalable" systems on the open web. But for usual business applications its complexity outweigh its benefits, therefore the Devonfw proposes a more "pragmatic" approach to REST services.

## HTTP Methods

On the next table we compare the main differences between the "canonical" REST approach (or RESTful) and the Devonfw proposal.

Table 2.4. Usage of HTTP methods

HTTP Method	RESTful Meaning	Devonfw
GET	<a href="http://localhost:8080/UserManagement/rest/UserService/users/1">http://localhost:8080/UserManagement/rest/UserService/users/1</a> Get User of Id 1 (Read Only)(Read Only)	Read a single element.
PUT	<a href="http://localhost:8080/UserManagement/rest/UserService/users/2">http://localhost:8080/UserManagement/rest/UserService/users/2</a> Insert User with Id 2 (Idempotent)	Not used
POST	<a href="http://localhost:8080/UserManagement/rest/UserService/users/2">http://localhost:8080/UserManagement/rest/UserService/users/2</a> Update User with Id 2 (N/A)	Create or update an element in the collection. Search on an entity (parametrized post body) Bulk deletion.
DELETE	<a href="http://localhost:8080/UserManagement/rest/UserService/users/1">http://localhost:8080/UserManagement/rest/UserService/users/1</a> Delete User with Id 1 (Idempotent)	Delete an entity. Delete an entry collection (typically not supported)

Please consider these guidelines and rationales: \* We use POST on the collection URL for both create and update operations on an entity. This avoids pointless discussions in distinctions between PUT and POST and what to do if a "creation" contains an ID or if an "update" is missing the ID property. \* Hence, we do NOT use PUT but always use POST for write operations. As we always have a technical ID for each entity, we can simply distinguish create and update by the presence of the ID property. Here are important points to be considered:

- GET operations are read only and are safe.
- PUT and DELETE operations are idempotent means their result will always same no matter how many times these operations are invoked.
- PUT and POST operation are nearly same with the difference lying only in the result where PUT operation is idempotent and POST operation can cause different result.

## JAX-RS

For implementing REST services we use the [JAX-RS](#) standard. As an implementation we recommend [CXF](#). JAX-RS stands for JAVA API for RESTful Web Services. JAX-RS is a JAVA based programming language API and specification to provide support for created RESTful Webservices. JAX-RS makes heavy use of annotations available from Java to simplify development of JAVA based web services creation and deployment. It also provides supports for creating clients for RESTful web services.

- Specifications

S.no	Annotation	Description
1	@Path	Relative path of the resource class/method.
2	@GET	HTTP Get request, used to fetch resource.
3	@PUT	HTTP PUT request, used to create resource.
4	@POST	HTTP POST request, used to create/update resource.
5	@DELETE	HTTP DELETE request, used to delete resource.
6	@HEAD	HTTP HEAD request, used to get status of method availability.
7	@Produces	States the HTTP Response generated by web service, for example application/xml, text/html, application/json etc.
8	@Consumes	States the HTTP Request type, for example application/x-www-form-urlencoded to accept form data in HTTP body during POST request.
9	@PathParam	Binds the parameter passed to method to a value in path.
10	@QueryParam	Binds the parameter passed to method to a query parameter in path.
11	@MatrixParam	Binds the parameter passed to method to a HTTP matrix parameter in path.
12	@HeaderParam	Binds the parameter passed to method to a HTTP header.
13	@CookieParam	Binds the parameter passed to method to a Cookie.
14	@FormParam	Binds the parameter passed to method to a form value.
15	@DefaultValue	Assigns a default value to a parameter passed to method.

S.no	Annotation	Description
16	@Context	Context of the resource for example <code>HTTPRequest</code> as a context.

Following are the commonly used annotations to map a resource as a web service resource.

If you want to know more about why we have chosen these options see [this](#). For `JSON` bindings we use [Jackson](#) while [XML](#) binding works out-of-the-box with [JAXB](#). To implement a service you simply write a regular class and use JAX-RS annotations to annotate methods that shall be exposed as REST operations. Here is a simple example:

```

@Path("/tablemanagement")
@Named("TableManagementRestService")
public class TableManagementRestServiceImpl implements RestService {
    ...
    @Produces(MediaType.APPLICATION_JSON)
    @GET
    @Path("/table/{id}")
    @RolesAllowed(PermissionConstant.GET_TABLES)
    public TableBo getTable(@PathParam("id") String id) throws RestServiceException {
        Long idAsLong;
        if (id == null)
            throw new BadRequestException("missing id");
        try {
            idAsLong = Long.parseLong(id);
        } catch (NumberFormatException e) {
            throw new RestServiceException("id is not a number");
        } catch (NotFoundException e) {
            throw new RestServiceException("table not found");
        }
        return this.tableManagement.getTable(idAsLong);
    }
    ...
}
  
```

Here we can see a REST service for the [business component](#) `tablemanagement`. The method `getTable` can be accessed via HTTP GET (see `@GET`) under the URL path `tablemanagement/table/{id}` (see `@Path` annotations) where `{id}` is the ID of the requested table and will be extracted from the URL and provided as parameter `id` to the method `getTable`. It will return its result (`TableBo`) as JSON (see `@Produces`). As you can see it delegates to the [logic](#) component `tableManagement` that contains the actual business logic while the service itself only contains mapping code and general input validation. Further you can see the `@RolesAllowed` for [security](#). The REST service implementation is a regular CDI bean that can use [dependency injection](#).

### Note

With JAX-RS it is important to make sure that each service method is annotated with the proper HTTP method (`@GET`,`@POST`,etc.) to avoid unnecessary debugging. So you should take care not to forget to specify one of these annotations.

### JAX-RS Configuration

Starting from CXF 3.0.0 it is possible to enable the auto-discovery of JAX-RS roots and providers thus avoiding having to specify each service bean in the `beans-service.xml` file.

When the jaxrs server is instantiated all the scanned root and provider beans (beans annotated with javax.ws.rs.Path and javax.ws.rs.ext.Provider) are configured. The xml configuration still allows us to specify the root path for all endpoints.

```
<jaxrs:server id="CxfRestServices" address="/rest" />
```

[[getting-started-creating-rest-service\_http-status codes]] === HTTP Status Codes

Further we define how to use the HTTP status codes for REST services properly. In general the 4xx codes correspond to an error on the client side and the 5xx codes to an error on the server side.

Table 2.5. Usage of HTTP status codes

HTTP Code	Meaning	Response	Comment
200	OK	requested result	Result of successful GET
204	No Content	none	Result of successful POST, DELETE, or PUT (void return)
400	Bad Request	error details	The HTTP request is invalid (parse error, validation failed)
401	Unauthorized	none (security)	Authentication failed
403	Forbidden	none (security)	Authorization failed
404	Not found	none	Either the service URL is wrong or the requested resource does not exist
500	Server Error	error code, UUID	Internal server error occurred (used for all technical exceptions)

For more details about REST service design please consult the [RESTful cookbook](#).

[[getting-started-creating-rest-service\_rest-exception handling]] === REST Exception Handling For exceptions a service needs to have an exception facade that catches all exceptions and handles them by writing proper log messages and mapping them to a HTTP response with an according [HTTP status code](#). Therefore the OASP4J provides a generic solution via RestServiceExceptionFacade. You need to follow the [exception guide](#) so that it works out of the box because the facade needs to be able to distinguish between business and technical exceptions. You need to configure it in your beans-service.xml as following:

```
<jaxrs:server id="CxfRestServices" address="/rest">
  <jaxrs:providers>
    <bean class="io.oasp.module.rest.service.impl.RestServiceExceptionFacade"/>
    <!-- ... -->
  </jaxrs:providers>
  <!-- ... -->
</jaxrs:server>
```

Now your service may throw exceptions but the facade will automatically handle them for you.

## Metadata

OASP4J has support for the following metadata in REST service invocations:

Name	Description	Further information
Correlation ID	A unique identifier to associate different requests belonging to the same session / action	<a href="#">Logging guide</a>
Validation errors	Standardized format for a service to communicate validation errors to the client	Server-side validation is documented in the <a href="#">Validation guide</a> .  The protocol to communicate these validation errors to the client is discussed at <a href="https://github.com/oasp/oasp4j/issues/218">https://github.com/oasp/oasp4j/issues/218</a>
Pagination	Standardized format for a service to offer paginated access to a list of entities	Server-side support for pagination is documented in the <a href="#">Data-Access Layer Guide</a> .

[[getting-started-creating-rest-service\_recommendations-for rest requests and responses]] ===  
Recommendations for REST requests and responses

The OASP4J proposes, for simplicity, a deviation from the REST common pattern:

- Using POST for updates (instead of PUT)
- Using the payload for addressing resources on POST (instead of identifier on the URL)
- Using parametrized POST for searches

This use of REST will lead to simpler code both on client and on server. We discuss this use on the next points.

REST services are called via HTTP(S) URIs. As we mentioned at the beginning we distinguish between **collection** and **element** URIs:

- A collection URI is build from the rest service URI by appending the name of a collection. This is typically the name of an entity. Such URI identifies the entire collection of all elements of this type. Example:

```
https://mydomain.com/myapp/services/rest/mycomponent/myentity
```

- An element URI is build from a collection URI by appending an element ID. It identifies a single element (entity) within the collection. Example:

```
https://mydomain.com/myapp/services/rest/mycomponent/myentity/42
```

The following table specifies how to use the HTTP methods (verbs) for collection and element URIs properly (see [wikipedia](#)). For general design considerations beyond this documentation see the [API Design eBook](#).

[[getting-started-creating-rest-service\_unparameterized-loading of a single resource]] ===  
Unparameterized loading of a single resource

- **HTTP Method:** GET
- **URL example:** /products/123

For loading of a single resource, embed the identifier of the resource in the URL (for example /products/123).

The response contains the resource in JSON format, using a JSON object at the top-level, for example:

```
{  
  "name": "Steak",  
  "color": "brown"  
}
```

[[getting-started-creating-rest-service\_unparameterized-loading of a collection of resources]] ===  
Unparameterized loading of a collection of resources

- **HTTP Method:** GET
- **URL example:** /products

For loading of a collection of resources, make sure that the size of the collection can never exceed a reasonable maximum size. For parameterized loading (searching, pagination), see below.

The response contains the collection in JSON format, using a JSON object at the top-level, and the actual collection underneath a result key, for example:

```
{  
  "result": [  
    {  
      "name": "Steak",  
      "color": "brown"  
    },  
    {  
      "name": "Broccoli",  
      "color": "green"  
    }  
  ]  
}
```

Avoid returning JSON arrays at the top-level, to prevent CSRF attacks (see [https://www.owasp.org/index.php/OWASP\\_AJAX\\_Security\\_Guidelines](https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines))

[[getting-started-creating-rest-service\_saving-a resource]] === Saving a resource

- **HTTP Method:** POST
- **URL example:** /products

The resource will be passed via JSON in the request body. If updating an existing resource, include the resource's identifier in the JSON and not in the URL, in order to avoid ambiguity.

If saving was successful, an empty HTTP 204 response is generated.

If saving was unsuccessful, refer below for the format to return errors to the client.

[[getting-started-creating-rest-service\_parameterized-loading\_of\_a\_resource]] === Parameterized loading of a resource

- **HTTP Method:** POST
- **URL example:** /products/search

In order to differentiate from an unparameterized load, a special *subpath* (for example search) is introduced. The parameters are passed via JSON in the request body. An example of a simple, paginated search would be:

```
{  
    "status": "OPEN",  
    "pagination": {  
        "page": 2,  
        "size": 25  
    }  
}
```

The response contains the requested page of the collection in JSON format, using a JSON object at the top-level, the actual page underneath a result key, and additional pagination information underneath a pagination key, for example:

```
{  
    "pagination": {  
        "page": 2,  
        "size": 25,  
        "total": null  
    },  
    "result": [  
        {  
            "name": "Steak",  
            "color": "brown"  
        },  
        {  
            "name": "Broccoli",  
            "color": "green"  
        }  
    ]  
}
```

Compare the code needed on server side to accept this request:

```
@Path("/order")  
@POST  
public List<OrderCto> findOrders(OrderSearchCriteriaTo criteria) {  
    return this.salesManagement.findOrderCtos(criteria);  
}
```

With the equivalent code required if doing it the REST way by issuing a GET request:

```
@Path("/order")  
@GET  
public List<OrderCto> findOrders(@Context UriInfo info) {  
  
    RequestParameters parameters = RequestParameters.fromQuery(info);  
    OrderSearchCriteriaTo criteria = new OrderSearchCriteriaTo();  
    criteria.setTableId(parameters.get("tableId", Long.class, false));  
    criteria.setState(parameters.get("state", OrderState.class, false));  
    return this.salesManagement.findOrderCtos(criteria);  
}
```

## Pagination details

The client can choose to request a count of the total size of the collection, for example to calculate the total number of available pages. It does so, by specifying the `pagination.total` property with a value of `true`.

The service is free to honour this request. If it chooses to do so, it returns the total count as the `pagination.total` property in the response.

[[getting-started-creating-rest-service\_deletion-of a resource]] === Deletion of a resource

- **HTTP Method:** `DELETE`
- **URL example:** `/products/123`

For deletion of a single resource, embed the identifier of the resource in the URL (for example `/products/123`).

## Error results

The general format for returning an error to the client is as follows:

```
{  
  "message": "A human-readable message describing the error",  
  "code": "A code identifying the concrete error",  
  "uuid": "An identifier (generally the correlation id) to help identify corresponding requests in logs"  
}
```

If the error is caused by a failed validation of the entity, the above format is extended to also include the list of individual validation errors:

```
{  
  "message": "A human-readable message describing the error",  
  "code": "A code identifying the concrete error",  
  "uuid": "An identifier (generally the correlation id) to help identify corresponding requests in logs",  
  "errors": {  
    "property failing validation": [  
      "First error message on this property",  
      "Second error message on this property"  
    ],  
    // ....  
  }  
}
```

[[getting-started-creating-rest-service\_rest-media types]] === REST Media Types

The payload of a REST service can be in any format as REST by itself does not specify this. The most established ones that the OASPI4J recommends are [XML](#) and [JSON](#). Follow these links for further details and guidance how to use them properly. JAX-RS and CXF properly support these formats (`MediaType.APPLICATION_JSON` and `MediaType.APPLICATION_XML` can be specified for `@Produces` or `@Consumes`). Try to decide for a single format for all services if possible and NEVER mix different formats in a service.

In order to use [JSON](#) via [Jackson](#) with [CXF](#) you need to register the factory in your `beans-service.xml` and make CXF use it as following:

```
<jaxrs:server id="CxfRestServices" address="/rest">  
  <jaxrs:providers>
```

```
<bean class="org.codehaus.jackson.jaxrs.JacksonJsonProvider">
  <property name="mapper">
    <ref bean="ObjectMapperFactory"/>
  </property>
</bean>
<!-- ... -->
</jaxrs:providers>
<!-- ... -->
</jaxrs:server>

<bean id="ObjectMapperFactory" factory-bean="RestaurantObjectMapperFactory" factory-
method="createInstance"/>
```

## REST Testing

For testing REST services in general consult the [testing guide](#).

For manual testing REST services there are browser plugins:

- Firefox: [httprequester](#) (or [poster](#))
- Chrome: [postman](#) ([advanced-rest-client](#))

[[getting-started-creating-rest-service\_details-of soap]] == Details of SOAP

## JAX-WS

For building web-services with Java we use the [JAX-WS](#) standard. There are two approaches:

- code first
- contract first

Here is an example in case you define a code-first service. We define a regular interface to define the API of the service and annotate it with JAX-WS annotations:

```
@WebService
public interface TablemanagementWebService {

  @WebMethod
  @WebResult(name = "message")
  TableEto getTable(@WebParam(name = "id") String id);

}
```

And here is a simple implementation of the service:

```
@Named("TablemanagementWebService")
@WebService(endpointInterface =
"io.oasp.gastronomy.restaurant.tablemanagement.service.api.ws.TablemanagementWebService")
public class TablemanagementWebServiceImpl implements TablemanagementWebService {

  private Tablemanagement tableManagement;

  @Override
  public TableEto getTable(String id) {

    return this.tableManagement.findTable(id);
  }
}
```

Finally we have to register our service implementation in the spring configuration file beans-service.xml:

```
<jaxws:endpoint id="tableManagement" implementor="#TablemanagementWebService" address="/ws/
Tablemanagement/v1_0"/>
```

The implementor attribute references an existing bean with the ID TablemanagementWebService that corresponds to the @Named annotation of our implementation (see [dependency injection guide](#)). The address attribute defines the URL path of the service.

[[getting-started-creating-rest-service\_soap-custom mapping]] ===== SOAP Custom Mapping

In order to map custom [datatypes](#) or other types that do not follow the Java bean conventions, you need to write adapters for JAXB (see [XML](#)).

#### SOAP Testing

For testing SOAP services in general consult the [testing guide](#).

For testing SOAP services manually we strongly recommend [SoapUI](#).

### 2.21.4 Service Considerations

The term *service* is quite generic and therefore easily misunderstood. It is a unit exposing coherent functionality via a well-defined interface over a network. For the design of a service we consider the following aspects:

- **self-contained**

The entire API of the service shall be self-contained and have no dependencies on other parts of the application (other services, implementations, etc.).

- **idem-potent**

E.g. creation of the same master-data entity has no effect (no error)

- **loosely coupled**

Service consumers have minimum knowledge and dependencies on the service provider.

- **normalized**

complete, no redundancy, minimal

- **coarse-grained**

Service provides rather large operations (save entire entity or set of entities rather than individual attributes)

- **atomic**

Process individual entities (for processing large sets of data use a [batch](#) instead of a service)

- **simplicity**

avoid polymorphism, RPC methods with unique name per signature and no overloading, avoid attachments (consider separate download service), etc.

### 2.21.5 Security

Your services are the major entry point to your application. Hence security considerations are important here.

#### CSRF

A common security threat is [CSRF](#) for REST services. Therefore all REST operations that are performing modifications (PUT, POST, DELETE, etc. - all except GET) have to be secured against CSRF attacks.

---

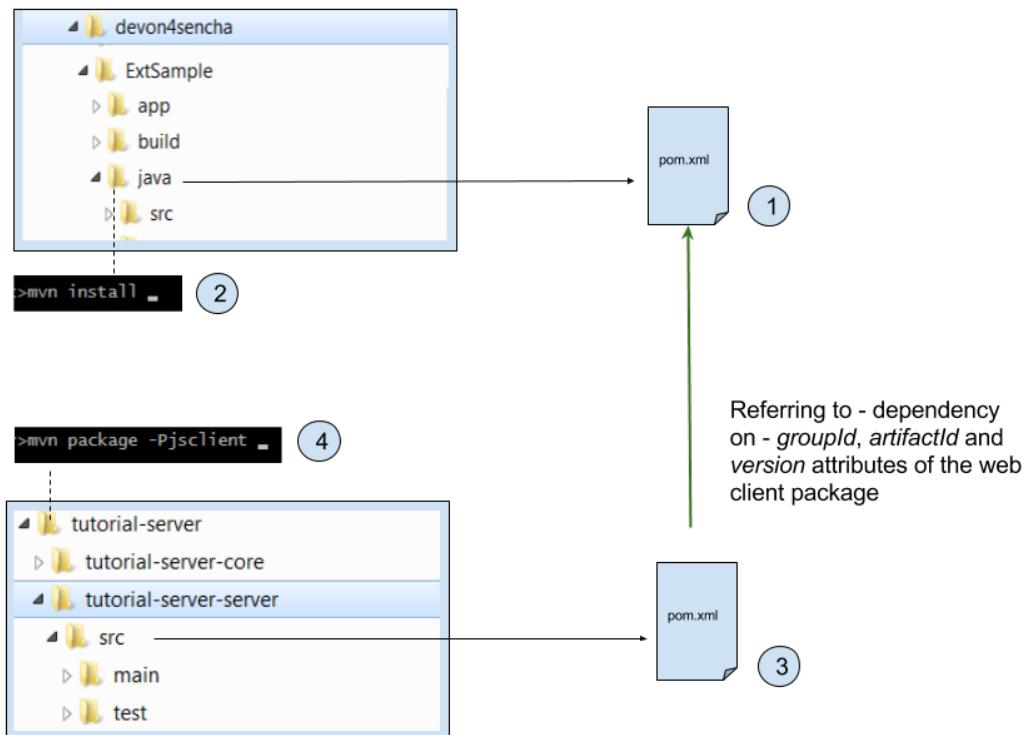
In OASP4J we are using spring-security that already solves CSRF token generation and verification. The integration is part of the application template as well as the sample-application.

For testing in development environment the CSRF protection can be disabled using the JVM option - DCsrfDisabled=true when starting the application.

## 2.22 Deployment on Tomcat (Client/Server)

After setting up functional server and client applications, we may want to package both in the same `.war` file. To package the single war, follow the given steps.

[[getting-started-deployment-on-tomcat\_general-description of the packaging process]] == General description of the packaging process The application packaging is based on *Maven package* functionality. The general overview of the packaging process is as follows:



[[getting-started-deployment-on-tomcat\_preparing-the client]] == Preparing the client

Firstly (1), both client applications (the Sencha and the Angular one) should contain a `java` directory with a `pom.xml` file which executes the build process (the "production" build, creating a single, compressed Javascript file from all the application files) through the command (2) `mvn install`. We must verify that the information about the `groupId`, the `artifactId` and the `version` are provided within the `pom.xml` file where we should find something like

```
...
<groupId>com.capgemini.devonfw</groupId>
<artifactId>ext js-sample</artifactId>
<version>1.0.0-SNAPSHOT</version>
...
```

So from the client application, in the `java` directory we launch the command

```
myClientApp\java>mvn install
```

After that, if the process goes right, the client app should have been "installed" in the local Maven repository of our environment so in the `\conf\m2\repository\com\capgemini\devonfw\extjs-sample\1.0.0-SNAPSHOT` directory we should find the `.jar` file with the client app packaged

Name	Type
 <code>_remote.repositories</code>	REPOSITORIES File
 <code>extjs-sample-1.0.0-SNAPSHOT.jar</code>	JAR File
 <code>extjs-sample-1.0.0-SNAPSHOT.pom</code>	POM File
 <code>extjs-sample-1.0.0-SNAPSHOT-web.zip</code>	zip Archive
 <code>maven-metadata-local.xml</code>	XML Document

[[getting-started-deployment-on-tomcat\_preparing-the server]] == Preparing the server The Java server application contains a `pom.xml` file (3). In this `pom.xml` file we should add the dependency to the `.jar` client that we have just created using the references to the `groupId`, `artifactId` and `version` that we have specified in the client `pom.xml`.

So in the `pom.xml` file of our server project we should add:

```
<dependency>
  <groupId>com.capgemini.devonfw</groupId>
  <artifactId>extjs-sample</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <type>zip</type>
  <classifier>web</classifier>
  <scope>runtime</scope>
</dependency>
```

And in the plugins of the `pom.xml` we should add a reference to the package again within the `<overlay>` tag:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <overlays>
      <overlay>
        <groupId>com.capgemini.devonfw</groupId>
        <artifactId>extjs-sample</artifactId>
        <type>zip</type>
        <classifier>web</classifier>
        <targetPath>jsclient</targetPath>
      </overlay>
    </overlays>
  </configuration>
</plugin>
```

**Note**

If you are using a Sencha project as client app you must comment all the `<execution>` tags from the `exec-maven-plugin` inside the `jsclient` profile as this configuration is related to `oasp4js` projects.

Now verify that the server redirects to the client checking the ...\\MyServerApp\\server\\src\\main\\webapp\\index.jsp file that should be

**index.jsp.**

```
<%  
    response.sendRedirect(request.getContextPath() + "/jsclient/");  
%>
```

Then we have to add some unsecured resources in the method `configure(HttpSecurity http)` of the `general/service/impl/config/BaseWebSecurityConfig.java` class.

Edit the `unsecureResources` to have something like that:

```
@Override  
public void configure(HttpSecurity http) throws Exception {  
  
    String[] unsecuredResources =  
        new String[] { "/login", "/security/**", "/services/rest/login", "/services/rest/logout", "/  
jsclient/**" };  
  
    (...)  
}
```

[[getting-started-deployment-on-tomcat\_packaging-the apps]] == Packaging the apps Finally we are going to package both client and server applications into the same .war file. To do that we must execute the `package` Maven command (4) from within the projects root directory (the parent of the server project).

```
mvn package -P jsclient
```

[[getting-started-deployment-on-tomcat\_deploy-on tomcat]] == Deploy on Tomcat

To deploy packaged Web Application Archive (.war) file that is integrated with client (Angular or Sencha Client) on Tomcat7/Tomcat 8, make below changes in java core application pom.xml file.

Example: For "oasp4j" project, make following changes in core application's "pom.xml" which is located in "oasp4j/samples/core/pom.xml".

- Modify dependency "spring-boot-starter-web" and add exclusions.
- Add new dependency "spring-boot-starter-tomcat".

```
...  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    <exclusions>  
        <exclusion>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-tomcat</artifactId>  
        </exclusion>  
    </exclusions>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
...
```

- Comment the code inside core\src\main\java\io\oasp\gastronomy\restaurant\general\service\impl\config\ServletInitializer.java. This is not needed as we will be overriding the 'configure' method inside core\src\main\java\io\oasp\gastronomy\restaurant\SpringBootApp.java.

```
public class SpringBootApp extends SpringApplication {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootApp.class);
    }

    /**
     * Entry point for spring-boot based app
     *
     * @param args - arguments
     */
    public static void main(String[] args) {
        SpringApplication.run(SpringBootApp.class, args);
    }
}
```

- Activate the 'jsclient' profile in server/pom.xml. Please see the snippet below.

```
<profiles>
    <profile>
        <id>jsclient</id>
        <activation>
            <!--<activeByDefault>true</activeByDefault>-->
            <activeByDefault>true</activeByDefault>
        </activation>
        .....
        .....
    </profile>
</profiles>
```

Build the project and create packaged .war file.

To deploy this .war file on Tomcat 7, follow the steps given below:

1. Go to Tomcat installation folder (TOMCAT\_HOME) → Copy .war file to "TOMCAT\_HOME/webapps" folder .
2. If Tomcat is running, stop it by running "shutdown.bat" file under "TOMCAT\_HOME/bin" folder.
3. Delete "TOMCAT\_HOME/temp" and "TOMCAT\_HOME/work" folders if present. These folders contain temporary files. (Mandatory to get desired output)
4. Start Tomcat by running "startup.bat" under "TOMCAT\_HOME/bin" folder.
5. By default Tomcat will start on port "8080".

[[getting-started-deployment-on-tomcat\_running-bootified war]] == Running Bootified War

To run bootified war file , follow the steps given below:

1. cd oasp4j\samples
2. Execute 'mvn clean install'
3. cd oasp4j\samples\server\target.
4. Execute 'java -jar oasp4j-sample-server-bootified.war'

[[getting-started-deployment-on-tomcat\_application-context root]] === Application context root In the case of bootified war, the context root will be '/' and not 'oasp4j-sample-server'.

So, to access the application after the bootified war is launched , one has to access it via <http://localhost:8080/login> or if the user wants to have a context root , then they can define the context 'oasp4j-sample-server' in oasp4j\samples\core\src\main\resources\application.properties

Make sure oasp4j\samples is built by executing 'mvn clean install' for this oasp4j\samples project and access it via <http://localhost:8080/oasp4j-sample-server>. The context root defined in oasp4j\samples\core\src\main\resources\config\application.properties will not be available since it is excluded from the war that is generated.

## 3. Devonfw CookBook

### 3.1 The Reporting module - Report generation with JasperReports

Reporting is a fundamental part of the larger movement towards the improved business intelligence and knowledge management. Often, the implementation involves extract, transform, and load (ETL) procedures in coordination with a data warehouse and then using one or more reporting tools. With this module, Devon provides an implementation of one of these reporting tools based on the Jasper Reports library.

JasperReports is an open source Java reporting tool that can write to a variety of targets, such as: screen, a printer, into PDF, HTML, Microsoft Excel, RTF, ODT, Comma-separated values or XML files. It can be used in Java-enabled applications, including Java EE or web applications, to generate dynamic content. It reads its instructions from an XML or .jasper file.

For more information, visit [JasperReports](#)

[[cookbook-reporting-module\_include-reporting in a devon project]] == Include Reporting in a Devon project

The Reporting module provides you a report generator for your Devon applications. To implement the Reporting module in a Devon project, you must follow these steps:

[[cookbook-reporting-module\_step-1: adding the dependency in your project]] === Step 1: Adding the dependency in your project

Include the module dependency in your pom.xml, verify that the *version* matches the last available version of the module.

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-reporting</artifactId>
  <version>2.2.0</version>
</dependency>
```

#### Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case, you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command *mvn install*. If the project is already imported in Eclipse, then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-reporting-module\_step-2: properties configuration]] === Step 2: Properties configuration

#### Note

This step is only needed in case you are going to generate *.txt* reports.

In order to use the Reporting module for creating txt reports, it is necessary to define some parameters related to the size of the elements in the *application.properties* file in the project.

```
# Reporting module params
devon.reporting.txtConfig.CharWidth=7
devon.reporting.txtConfig.CharHeight=13.9
devon.reporting.txtConfig.PageWidthInChars=80
devon.reporting.txtConfig.PageHeightInChars=47
```

### 3.1.1 Basic implementation

First and foremost, you need to add the scanner for dependency injection. To do so, you must add the following annotations in the *SpringBoot* main class:

```
@Configuration
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.reporting" })
@EnableAutoConfiguration
public class MyBootApp {

    [...]
}
```

Remember to include the package of the module in the *basePackages* attribute of the `@ComponentScan` annotation alongside the packages for the rest of the relevant Spring Boot components.

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.reporting",
    "my.other.component.location.package" })
```

As you can see, the *basePackages* of the `@ComponentScan` points to the Reporting module package. Now, you can start using the module.

[[cookbook-reporting-module\_the-injection of the module]] === The injection of the module

To access the module functionalities, you need to inject the module in a private property, it can be done using the `@Inject` annotation

```
public class MyClass {

    @Inject
    private Reporting<Map> reportManager;

    [...]
}
```

Hereafter, you can use the `reportManager` object in order to access the module functionalities, it will be discussed later.

[[cookbook-reporting-module\_the-report entity]] === The Report entity

Basically, for configuring the report, you need to instantiate a `Report` object and define only two properties:

- the data: the information that the report should show.
- the template: the `.jrxml` file that the report engine will use to format, order and set the style of the report.

```
Report myReport = new Report();

myReport.setData(getMockData());
myReport.setTemplatePath("path\to\the\template\fooTemplate.jrxml");
```

The `setData` method needs the **collection of `HashMap`** with the pairs key/value to bind template fields with the data.

In the `setTemplatePath`, you need to pass the location of the template which will be used to create the report. You can learn more about how to create Jasper templates [here](#) and [here](#).

In the `Report` object, you can also add parameters that can be used within the template:

```
HashMap<String, Object> params = new HashMap<>();
params.put("ReportTitle", "Foo");
params.put("ReportDescription", "Report generated using the Devon Reporting module");

myReport.setParams(params);
```

[[cookbook-reporting-module\_using-the reportmanager]] == Using the reportManager

Once the `Report` object is defined and configured, you can generate the report. Following example shows a basic implementation for the creation of a report in pdf file

```
File file = new File("D:\\Temp\\pdf_Report.pdf");
reportManager.generateReport(myReport, file, ReportFormat.PDF);
```

Therefore, once the `Report` object is defined, the report generation is very simple, it only needs:

- a report manager (the object with the injection of the module).
- the `Report` object with the `data` and the `template` defined.
- a file to *write* the report results.
- a format for the report (you can choose between pdf, xls, xlsx, doc, docx, txt, html, Pptx and several more).

[[cookbook-reporting-module\_working-with templates]] == Working with templates

With reference to previous sections, the Reporting module works using the *Jasper Reports* templates. These templates are basically *xml* files (with extension *jxml*) with some custom structure.

[[cookbook-reporting-module\_the-parts of a template]] == The parts of a template

The *jxml* templates are divided into several blocks of information. These blocks can be of two types:

- blocks with static information.
- blocks with dynamic information.

The static information is the information defined by the template itself or by the parameters passed to the template and it remains unchanged over the different *pages* of the report.

The dynamic information is the information defined by the `data` that is passed to the `Report` object as it is the report's main content.

A basic *jxml* structure would be like below:

```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge.n.....">
  <parameter .... />
  <parameter .... />
  <field .... />
  <field .... />
  <field .... />
```

```
<title> [...] </title>  
  
<pageHeader> [...] </pageHeader>  
  
<columnHeader> [...] </columnHeader>  
  
<detail> [...] </detail>  
  
<columnFooter> [...] </columnFooter>  
  
<pageFooter> [...] </pageFooter>  
  
<summary> [...] </summary>  
  
</jasperReport>
```

- **title** tag: will store static information and will appear only once on the first page of the report.
- **pageHeader** tag: will contain static information and will appear on every report page at the top of the page.
- **columnHeader** tag: will show static information and will appear on every report page, just above the *detail* info.
- **detail** tag: will contain the dynamic content of the report and will be repeated (in row format) many times as the occurrence of the data that is passed in the *setData* method. The detail will fill the page report and continue in the following pages if is necessary.
- **columnFooter** tag: will show static information and will appear on the every report page, just below the *detail* info, at the end of the *detail* info gap in every report page.
- **pageFooter** tag: will contain static information and will appear on every report page in the bottom of the page.

## Defining parameters

Parameters in the templates can be defined in this way and after the `<jasperReport>` tag.

```
<parameter name="ReportTitle" class="java.lang.String"/>  
<parameter name="ReportDescription" class="java.lang.String"/>
```

[[cookbook-reporting-module\_using-parameters in the template]] === Using parameters in the template

After the parameter definition, you can use the parameters within the template with a structure shown below:

```
<textField>  
  <reportElement .... />  
  <textElement>  
  </textElement>  
  <textFieldExpression><![CDATA[$P{ReportTitle}]]&gt;&lt;/textFieldExpression&gt;<br/></textField>
```

## Defining Fields

The fields are the elements linked with the reports dynamic data. The fields can be defined in the templates in this way and after the `<jasperReport>` tag.

```
<field name="ID" class="java.lang.Integer"/>  
<field name="Name" class="java.lang.String"/>
```

[[cookbook-reporting-module\_using-fields in the template]] === Using fields in the template

After the field definition, you can use the fields inside the `<detail>` tag as the part of the dynamic data.

```
[...]  
  
<detail>  
  <band .... >  
    <line>  
      <reportElement .... />  
    </line>  
    <textField .... >  
      <reportElement .... />  
      <textElement>  
        <font size= .... />  
      </textElement>  
      <textFieldExpression class="java.lang.Integer"><![CDATA[$F{ID}]></textFieldExpression>  
    </textField>  
    <textField .... >  
      <reportElement .... />  
      <textElement>  
        <font size= .... />  
      </textElement>  
      <textFieldExpression class="java.lang.String"><![CDATA[$F{Name}]></textFieldExpression>  
    </textField>  
  
[...]
```

[[cookbook-reporting-module\_creating-templates with gui software]] === Creating templates with GUI software

Working with *xml* can be sometimes complex and it adds a layer of difficulty when trying to visualize a graphic result. For that reason, Jaspersoft provides a software to manage the Reports and this software includes a complete functionality to generate and export *jxml* templates.

It is about Jaspersoft Studio and you can get it from the Jaspersoft site [here](#).

In the similar way, the Jaspersoft site provides the users with many documentation and examples of how to use Jaspersoft studio, how to install it and how to generate templates:

- [Getting Started with Jaspersoft Studio](#)
- [Designing a Report with Jaspersoft Studio](#)
- [Creating a custom template with Jaspersoft Studio](#)

### 3.1.2 Subreports

A subreport is a report included inside another report. This allows the creation of very complex layouts with different portions of a single document filled using different data sources and reports. To know more about subreports, refer this [link](#).

A basic example of the subreports usage with the Reporting module is below:

```
File file = File.createTempFile("subreport_", ".pdf");  
this.reportManager.generateSubreport(masterReport, subreports, file, ReportFormat.PDF);
```

- The *masterReport* is the report that will house the sub-reports. It is defined as it is explained in the previous section.
- The *subreports* is a List of reports to be included within the main report.
- The rest of parameters are explained in the previous section.

[[cookbook-reporting-module\_defining-a subreport]] ===== Defining a Subreport

The subreport definition is same as for a regular report, the only point is to define the `setDataSourceName`.

```
List<Report> subreports = new ArrayList<>();  
[...]  
Report sureport01 = new Report();  
sureport01.setName("subreport01");  
sureport01.setDataSourceName("subreport01DataSource");  
sureport01.setSubreport01MockData();  
sureport01.setTemplatePath(path\to\the\template\sureport01Template.jrxml);  
this.subreports.add(sureport01);
```

The `DataSourceName` is the name, that will be later used to bind the subreport with its data, so that it has to be defined in the master report template in order to pass it to the subreport as a parameter.

```
[...]  
  
<parameter name="subreport01" class="net.sf.jasperreports.engine.JasperReport"/>  
<parameter name="subreport01DataSource" class="net.sf.jasperreports.engine.JRDataSource" />  
  
[...]  
  
<subreport>  
    <reportElement ... />  
    <dataSourceExpression><![CDATA[$P{subreport01}]]></dataSourceExpression>  
    <subreportExpression><![CDATA[$P{subreport01DataSource}]]></subreportExpression>  
</subreport>
```

[[cookbook-reporting-module\_how-to pass a parameter to a subreport]] ===== How to pass a parameter to a subreport

You can pass a parameter to a subreport using the `setParams` method of the master report.

```
// You will have a HashMap for "global" parameters  
HashMap<String, Object> allParams = new HashMap<>();
```

Then, when defining a subreport, you can add its parameters to the *global* parameters:

```
HashMap<String, Object> subreport01Params = new HashMap<>();  
subreport01Params.put("City", "Valencia");  
allParams.putAll(subreport01Params);
```

And during the master report definition:

```
this.masterReport.setParams(allParams);
```

Finally, in the master report template, you will define the parameter and pass it to the subreport.

```
[...]  
  
<parameter name="City" class="java.lang.String" />  
  
[...]  
  
<subreport>  
    <reportElement ... />  
    <subreportParameter name="City">  
        <subreportParameterExpression><![CDATA[$P{City}]]></subreportParameterExpression>  
    </subreportParameter>  
    <dataSourceExpression ... />  
    <subreportExpression ... />  
</subreport>
```

## Concatenated reports

Other functionality of the Reporting module is to generate concatenated reports. A concatenated report is a set of reports *printed* in a single file. In other words, you can have several reports and generate a single file to contain them all.

A basic example of this:

```
this.reportManager.concatenateReports(reports, file, ReportFormat.PDF);
```

The *reports* parameter is a List of *Report* objects. The rest of the parameters are same as explained in the previous sections.

## 3.2 The Winauth-AD module

This Devonfw IP module allows the applications to authenticate the users against an Active Directory.

[[cookbook-winauth-ad-module\_authentication-with active directory]] == Authentication with Active Directory

Active Directory (AD) is a directory service that Microsoft developed for Windows domain networks. It is included in the most Windows Server operating systems as a set of processes and services. Initially, Active Directory was only in charge of centralized domain management. Starting with Windows Server 2008, however, Active Directory became an umbrella title for a broad range of directory-based identity-related services.

For more information, visit [wikipedia](#).

[[cookbook-winauth-ad-module\_include-winauth-ad in a devon project]] === Include Winauth-ad in a Devon project

Winauth-ad module provides a simple authentication for your Devon applications. To implement authentication in your Devon project, follow the next steps:

[[cookbook-winauth-ad-module\_step-1: add the module dependency]] ===== Step 1: Add the module dependency

Include the dependency of the module in pom.xml. Verify that the *version* matches the last available version of the module.

```
<dependency>
    <groupId>com.capgemini.devonfw.modules</groupId>
    <artifactId>devonfw-winauth-ad</artifactId>
    <version>2.2.0</version>
</dependency>
```

### Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command `mvn install`. If project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-winauth-ad-module\_step-2: security configuration ]] ===== Step 2: Security configuration

Create a variable of the class *AuthenticationManagerAD* in *general/service/impl/config/BaseWebSecurityConfig.java*

```
@Inject
private AuthenticationManagerAD authenticationManagerAD;
```

### Note

For previous versions of the *oasp4j* based apps, you may find *BaseWebSecurityConfig* in a different location: *general/configuration/BaseWebSecurityConfig.java*

Remember to add the package of the module to the `@ComponentScan` annotation in the Spring Boot main class.

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.winauthad" , "my.other.components.package" })
```

[[cookbook-winauth-ad-module\_step-3: define the provider]] ===== Step 3: Define the provider

Also, in the `BaseWebSecurityConfig.java` class, add the LDAP provider to the `AuthenticationManagerBuilder` in the `configureGlobal(AuthenticationManagerBuilder auth)` method

```
auth.authenticationProvider(this.authenticationManagerAD.LdapAuthenticationProvider());
```

### Note

For previous version of `oasp4j` apps, you won't find 'configureGlobal' method. Therefore, you should add it in the `init()` method instead (in such cases, you should find an `AuthenticationManagerBuilder` available in the class).

```
@PostConstruct
public void init() throws Exception {
    this.authenticationManagerBuilder
        .authenticationProvider(this.authenticationManagerAD.LdapAuthenticationProvider());
}
```

[[cookbook-winauth-ad-module\_step-4: implement the.userdetailscontextmapper]] ===== Step 4: Implement the `UserDetailsContextMapper`

Implement the class `UserDetailsContextMapper` to build the `UserDetails` with the data of the user.

```
import com.capgemini.devonfw.module.winauthad.common.api.AuthenticationSource;
import com.capgemini.devonfw.module.winauthad.common.api.UserData;
import com.capgemini.devonfw.module.winauthad.common.impl.security.GroupMapperAD;
import com.capgemini.devonfw.module.winauthad.common.impl.security.PrincipalProfileImpl;

@Named
@Component
public class UserDetailsContextMapperImpl implements UserDetailsContextMapper {

    private static final Logger LOG = LoggerFactory.getLogger(UserDetailsContextMapperImpl.class);

    @Inject
    private AuthenticationSource authenticationSource;

    @Inject
    private GroupMapperAD groupMapperAD;

    @Inject
    private AccessControlProvider accessControlProvider;

    /**
     * @return authenticationSource
     */
    public AuthenticationSource getAuthenticationSource() {
        return this.authenticationSource;
    }

    /**
     * @param authenticationSource new value of authenticationSource.
     */
    public void setAuthenticationSource(AuthenticationSource authenticationSource) {
```

```
        this.authenticationSource = authenticationSource;
    }

    /**
     * @param accessControlProvider new value of accessControlProvider.
     */
    public void setAccessControlProvider(AccessControlProvider accessControlProvider) {

        this.accessControlProvider = accessControlProvider;
    }

    /**
     * @return groupMapperAD
     */
    public GroupMapperAD getGroupMapperAD() {

        return this.groupMapperAD;
    }

    /**
     * @param groupMapperAD new value of groupMapperAD.
     */
    public void setGroupMapperAD(GroupMapperAD groupMapperAD) {

        this.groupMapperAD = groupMapperAD;
    }

    @Override
    public UserDetails mapUserFromContext(DirContextOperations ctx, String username,
                                           Collection<? extends GrantedAuthority> authorities) {

        UserData user = new UserData(username, "", authorities);

        try {
            Attributes attributes = this.authenticationSource.searchUserByUsername(username);

            String cn = attributes.get("cn").toString().substring(4); // Username
            String givenname = attributes.get("givenname").toString().substring(11); // FirstName
            String sn = attributes.get("sn").toString().substring(4); // LastName
            String memberOf = attributes.get("memberof").toString().substring(10); // Groups

            PrincipalProfileImpl userProfile = new PrincipalProfileImpl();
            userProfile.setName(cn);
            userProfile.setFirstName(givenname);
            userProfile.setLastName(sn);
            userProfile.setId(cn);
            ArrayList<String> groups = this.groupMapperAD.groupsMapping(memberOf);

            userProfile.setGroups(groups);

            // determine granted authorities for spring-security...
            Set<GrantedAuthority> authoritiesAD = new HashSet<>();
            Collection<String> accessControlIds = groups;
            Set<AccessControl> accessControlSet = new HashSet<>();
            for (String id : accessControlIds) {
                boolean success = this.accessControlProvider.collectAccessControls(id, accessControlSet);
                if (!success) {
                    LOG.warn("Undefined access control {}.", id);
                    // authorities.add(new SimpleGrantedAuthority(id));
                }
            }
            for (AccessControl accessControl : accessControlSet) {
                authoritiesAD.add(new AccessControlGrantedAuthority(accessControl));
            }

            user = new UserData(username, "", authoritiesAD);
            user.setUserProfile(userProfile);
        } catch (Exception e) {
            e.printStackTrace();
            UsernameNotFoundException exception = new UsernameNotFoundException("Authentication failed.", e);
        }
    }
}
```

```
LOG.warn("Failed com.capgemini.devonfw.module.winauthad.common.impl.security get user {} in Active
Directory."
        + username + exception);
    throw exception;
}
return user;
}

@Override
public void mapUserToContext(UserDetails user, DirContextAdapter ctx) {
}
}
```

### Note

Therefore, the above code builds the user with the Active Directive information. And the map of the groups in the configuration.

You can build any User you want. For e.g. you could use a query to Active Directory (like the example) or a query to your own User database.

[[cookbook-winauth-ad-module\_step-5: configure the ldap-ad connection]] ===== Step 5: Configure the LDAP-AD connection

Now, you need to configure the LDAP parameters in *application.properties*. By default, the *winauth-ad* module works with a LDAP Authentication and a query to AD to have the authorization, so you need to define all these properties. If you are using a customized *UserDetails* without AD query, you don't need to define the AD properties. The same happens, if you don't use the *Role Mapping* class.

```
#Server configuration
#LDAP
devon.winauth.ldap.url=ldap://mydomain.com/
devon.winauth.ldap.encrypt=true
devon.winauth.ldap.keyPass=keyPass
devon.winauth.ldap.password=ENC(...)
devon.winauth.ldap.userDn=cn=user,DC=mydomain,DC=com
devon.winauth.ldap.patterns=ou=Users
devon.winauth.ldap.userSearchFilter=(sAMAccountName={0})
devon.winauth.ldap.userSearchBase=

#AD
devon.winauth.ad.url=ldap://mydomain.com/OU=Users,DC=MYDOMAIN,DC=COM
devon.winauth.ad.domain=mydomain.com
devon.winauth.ad.username=user
devon.winauth.ad.encrypt=true
devon.winauth.ad.keyPass=keyPass
devon.winauth.ad.password=ENC(...)
devon.winauth.ad.userSearchFilter=(uid={0})
devon.winauth.ad.userSearchBase=
devon.winauth.ad.searchBy=sAMAccountName
devon.winauth.ad.rolePrefix=^(.*)CN=([^\,]*),.*,DC=MYDOMAIN,DC=COM$

#Roles mapping
devon.winauth.groups.Chief=S-ESPLAN
devon.winauth.groups.Waiter=S-ECOMU7
devon.winauth.groups.Cook=dlescapgemini.grado-a
devon.winauth.groups.TESTGROUP=testGroup
```

Now you can run your application and show the login form with the Active Directory authentication.

**Note**

As you can see the property password is encrypt. You can find more information about it [here](#).  
Also you can put the password without encrypt by default.

[[cookbook-winauth-ad-module\_using-the userdetailscontextmapper with ad]] === Using the UserDetailsContextMapper with AD

As mentioned above, you can implement your own *UserDetailsContextMapper* or use the *UserDetailsContextMapper* given in this tutorial. If you use the last one, you need to keep in mind the next points.

[[cookbook-winauth-ad-module\_roler-and groups mapper]] ===== Roler and Groups mapper

*Winauth-ad* includes a group mapper that gives a simple tool to map the groups of the Active Directory with a roles/groups of your application. To use it, you need to configure the mapping as shown below:

```
#Roles mapping
devon.winauth.groups.SESPLAN=S-ESPLAN
devon.winauth.groups.ECOMU7=S-ECOMU7
devon.winauth.groups.GradoA=dlescapgemini.grado-a
devon.winauth.groups.TESTGROUP=testGroup
```

Now, if you ask the server for the current user of the application, you will see the user data with his groups.

### 3.2.1 Service CurrentUser

If you use the basic *UserDetailsContextMapper* that *winauth-ad* implements, you need to modify the service *currentuser* in the class *general/service/impl/rest/SecurityRestServiceImpl.java*.

```
@Produces(MediaType.APPLICATION_JSON)
@GET
@Path("/currentuser/")
@PermitAll
public UserDetailsClientToAD getCurrentUser(@Context HttpServletRequest request) {

    if (request.getRemoteUser() == null) {
        throw new NoActiveUserException();
    }
    return UserData.get().toClientTo();
}
```

**Note**

You need to *import* the classes *UserData* and *UserDetailsClientToAD* of the *winauth-ad* module.

### 3.3 The Winauth-SSO module

This Devonfw IP module allows the applications to authenticate the users using the Windows credentials. The basic result is that the login is made automatically through the browser avoiding the login form (in some browsers like *Firefox*, you may need to authenticate once).

[[cookbook-winauth-sso-module\_single-sign on]] == Single Sign On

Single sign-on (SSO) is a property of access control of multiple related, but independent software systems. With this property, a user logs in with a single ID and password to gain access to a connected system or systems without using different usernames or passwords, or in some configurations seamlessly sign on at each system.

For more information, visit [wikipedia](#).

[[cookbook-winauth-sso-module\_include-winauth sso in a devon project]] === Include Winauth SSO in a Devon project

*Winauth* SSO module provides a simple *Single sign-on* authentication for your Devon applications. If you want to implement this kind of authentication in a Devon project, follow the next steps:

[[cookbook-winauth-sso-module\_step-1: add the dependency]] ===== Step 1: Add the dependency

Include the dependency of the module in pom.xml. Verify that the *version* matches the last available version of the module:

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-winauth-sso</artifactId>
  <version>2.2.0</version>
</dependency>
```

#### Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command *mvn install*. If the project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-winauth-sso-module\_step-2: inject the module]] ===== Step 2: Inject the module

The class *general/service/impl/config/BaseWebSecurityConfig.java* uses the *@Inject* annotation to get access to the module through a private field.

```
import com.capgemini.devonfw.module.winauthsso.common.api.WinauthSSO;
{...}
public abstract class BaseWebSecurityConfig extends WebSecurityConfigurerAdapter {
{...}
@Inject
private WinauthSSO sso;
```

```
{...}  
}
```

### Note

For previous versions of the *oasp4j* based apps, you may find `BaseWebSecurityConfig` in a different location: `general/configuration/BaseWebSecurityConfig.java`.

Remember to add the package of the module to the `@ComponentScan` annotation in the Spring Boot main class.

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.winauthsso"  
, "my.other.components.package" })
```

[[cookbook-winauth-sso-module\_step-3: define the security entry point and filter]] ===== Step 3: Define the security entry point and filter

Also, add the winauth SSO configuration down in the void `configure(HttpSecurity)` method in the `BaseWebSecurityConfig.java` class

```
@Override  
public void configure(HttpSecurity http) throws Exception {  
  
    //Winauth SSO configuration  
    http.addFilterAfter(this.sso.getSSOFilter(), BasicAuthenticationFilter.class).exceptionHandling()  
        .authenticationEntryPoint(this.sso.getSSOFilterEntryPoint());  
}
```

And that's all! Now, you have a simple SSO Authentication implemented.

Hereafter, when you access the app, the browser should redirect you to the main page of your app without the login process.

Henceforth, in the server console, you should see a message as shown below:

```
waffle.spring.NegotiateSecurityFilter : successfully logged in user: {Your-Domain}\{Your-User}
```

### Note

You need to be careful with the service's current user because SSO by default is not compatible with the information of the `UserDetailsClientTo` class. You need to adapt this class or use a customized SSO User Details (next chapter in the wiki).

[[cookbook-winauth-sso-module\_customized-winauth sso user details]] ===== Customized Winauth SSO User Details

According to the recent steps, you have a very simple authentication and authorization functionality with the Windows credentials. In the standard scenario, you may want to implement your own *User Details*. Therefore, let's discuss how to implement it for SSO authentication.

[[cookbook-winauth-sso-module\_step-1: create customized filter]] ===== Step 1: Create customized filter

The idea is to rebuild the default filter `NegotiateSecurityFilter`, you can create a complete new filter or, like this example, just modify some methods. In this case, you need to modify `boolean setAuthentication(...)`, this method is called by the method `void doFilter(...)` (you can

---

modify this method too) when the authentication is successful, so let's discuss how to build a custom *UserDetails*.

```
/**
 * This is a dummy implementation of a customized NegotiateSecurityFilter.
 *
 * @author jhcore
 */
public class NegotiateSecurityFilterCustomized extends NegotiateSecurityFilter {
    /** The Constant LOGGER. */
    private static final Logger LOGGER = LoggerFactory.getLogger(NegotiateSecurityFilterCustomized.class);

    private Usermanagement usermanagement = new UsermanagementDummyImpl();

    private AccessControlProvider accessControlProvider;

    /**
     * The constructor.
     *
     * @param accessControlProvider is the provider that help us to get the permissions
     */
    public NegotiateSecurityFilterCustomized(AccessControlProvider accessControlProvider) {
        super();
        this.accessControlProvider = accessControlProvider;
    }

    /**
     * The constructor.
     */
    public NegotiateSecurityFilterCustomized() {
        super();
    }

    @Override
    public void doFilter(final ServletRequest req, final ServletResponse res, final FilterChain chain)
        throws IOException, ServletException {
        // Here you can customize your own filter functionality
        super.doFilter(req, res, chain);
    }

    @Override
    protected boolean setAuthentication(final HttpServletRequest request, final HttpServletResponse response,
        final Authentication authentication) {
        try {
            String principal[] = authentication.getPrincipal().toString().split("\\\\", 2);

            String username = principal[1];

            UserProfile profile = this.usermanagement.findUserProfileByLogin(username);

            UsernamePasswordAuthenticationToken auth =
                new UsernamePasswordAuthenticationToken(profile, getAuthoritiesByProfile(profile));

            SecurityContextHolder.getContext().setAuthentication(auth);
        } catch (Exception e) {
            NegotiateSecurityFilterCustomized.LOGGER.warn("error authenticating user");
            NegotiateSecurityFilterCustomized.LOGGER.trace("", e);
        }
        return true;
    }

    private Object getAuthoritiesByProfile(UserProfile profile) {
        Set<GrantedAuthority> authorities = new HashSet<>();
        Collection<String> accessControlIds = new ArrayList<>();
        accessControlIds.add(profile.getRole().getName());
        Set<AccessControl> accessControlSet = new HashSet<>();
        for (String id : accessControlIds) {
            boolean success = this.accessControlProvider.collectAccessControls(id, accessControlSet);
        }
    }
}
```

```
        if (!success) {
            // authorities.add(new SimpleGrantedAuthority(id));
        }
    }
    for (AccessControl accessControl : accessControlSet) {
        authorities.add(new AccessControlGrantedAuthority(accessControl));
    }
    return authorities;
}
}
```

The above example uses the `UsermanagementDummyImpl`, which is generated during the creation of the new Devon application. Feel free to customize your own filter, just use the above class with a customized `Usermanagement`.

[[cookbook-winauth-sso-module\_step-2: inject and configure winauth sso]] ===== Step 2: Inject and configure Winauth SSO

Now, let's discuss how to create a Winauth SSO variable and to configure the filter.

```
import com.capgemini.devonfw.module.winauthsso.common.api.WinauthSSO;
{...}

public abstract class BaseWebSecurityConfig extends WebSecurityConfigurerAdapter {

    {...}

    @Inject
    private WinauthSSO sso;

    @Bean
    public AccessControlProvider accessControlProvider() {

        return new AccessControlProviderImpl();
    }

    {...}
}
```

As shown above, the Filter needs a `AccessControlProvider`, there is a one which is configured in the `WebSecurityConfig`, so you just need to pass it to the filter by param.

[[cookbook-winauth-sso-module\_step-3: configure the custom filter and the security entry point]] ===== Step 3: Configure the Custom Filter and the security entry point

Add the `winauth SSO` configuration down in the void `configure(HttpSecurity)` method

```
@Override
public void configure(HttpSecurity http) throws Exception {
    ...

    // Set the custom filter
    this.sso.setCustomFilter(new NegotiateSecurityFilterCustomized(accessControlProvider()));

    // Add the Filter to the app authentication process
    http.addFilterAfter(this.sso.getSSOFilter(), BasicAuthenticationFilter.class).exceptionHandling()
        .authenticationEntryPoint(this.sso.getSSOFilterEntryPoint());
}
```

And that's all! Now, you have a simple SSO Authentication with a custom `UserDetails` and you can use the server `current user` by default.

## 3.4 The i18n module

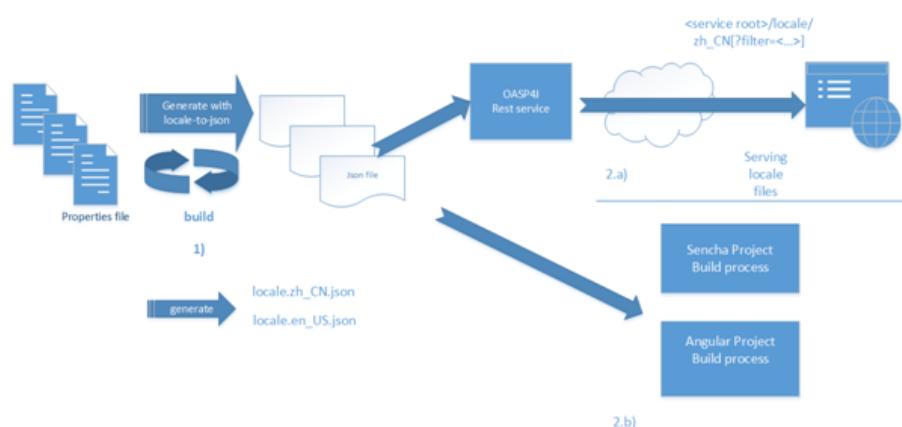
### 3.4.1 Introduction

Internationalization is the process of designing a software application so that it can potentially be adapted to specific local languages and regions without engineering changes. This constitutes much more than just "translating text or correct number formats". Think of internationalization as readiness for localization. Sometimes written as "i18n", internationalization evolved from a growing demand for multilingual products and applications.

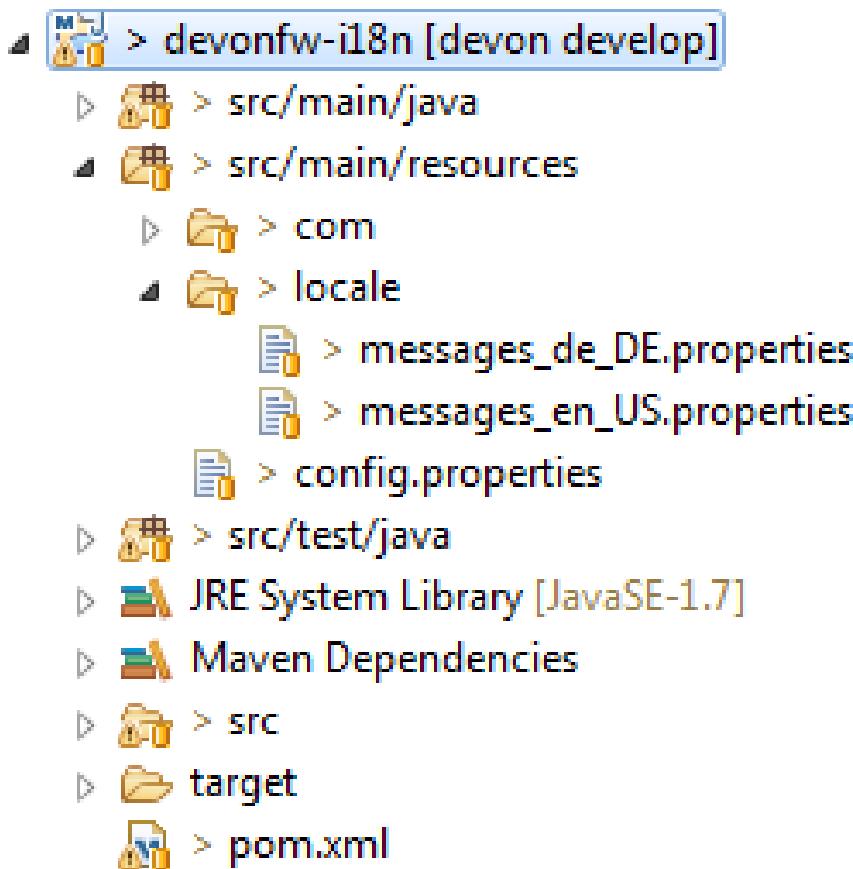
[[cookbook-i18n-module\_the-i18n ip module]] == The i18n IP Module

The i18n module provides easy creation and maintenance of distinct translations for your Devon applications. OASP, Angular and Sencha - three distinct platforms currently forming Devon. Each provides their own specific mechanisms for L10N. I18n module fits within the context of the facilities offered by each platform. I18n artifacts are available at compile/build time and not just "run-time". It is based on a namespaced name-value system ("properties").

#### Conceptual schema



As shown in above diagram, spring applications will have properties file defined for respective locales. This file will contains the translations of the applications texts as well as display format of elements like dates, numbers etc. These should be places within the `src/main/resources/locale` folder/namespace as shown in below dig.



I18n will convert these files to json data which will be return from REST services. The json files will reflect in structure the "dot namespaces" from the property file. So: main.intro.name = Devon fw control panel Becomes

```
{"main":  
  {"intro":  
    {  
      "name": "Devon fw control panel"  
    }  
  }  
}
```

To retrieve the response in the above said JSON format, User should make the Rest Service call by hitting the rest service. Url will be in form

```
<service root>/locales/<locale>  
For example:  
http://localhost:8081/oasp4j-sample-server/services/rest/i18n/locales/en_US
```

Filters:

If user is interested in retrieving the value of the property, he can get the same by using the feature 'Filters'. For Example: <service root>/locale/en\_US?filter=main.info In the example above, key 'main.info' is supplied as filter. Rest Service retrieves the corresponding value for this key and the JSON format of the value is returned to the user.

## Exception behavior

In case of usage of filter and no data returned (no properties were found), an empty JSON response is returned denoting no results. In case of an invalid Locale, user will receive DevonfwUnknownLocaleException.

[[cookbook-i18n-module\_i18n-mechanism integration]] === I18N mechanism integration

Currently three libraries are available for server-side i18n. Those libraries are:

- mmm-util-core and mmm-util-cli
- Standard library (i18n module use com.google.code.gson to convert java property file to json)
- Customized library (User needs to develop this module)

In devonfw-i18n module, config.properties file contains 'i18n.input.name' property to denote library used for i18n. User needs to set module name in config.properties file. User can either set 'standard' or 'mmm' or customized module name.

[[cookbook-i18n-module\_mmm-util-core-and mmm-util-cli]] ===== mmm-util-core and mmm-util-cli

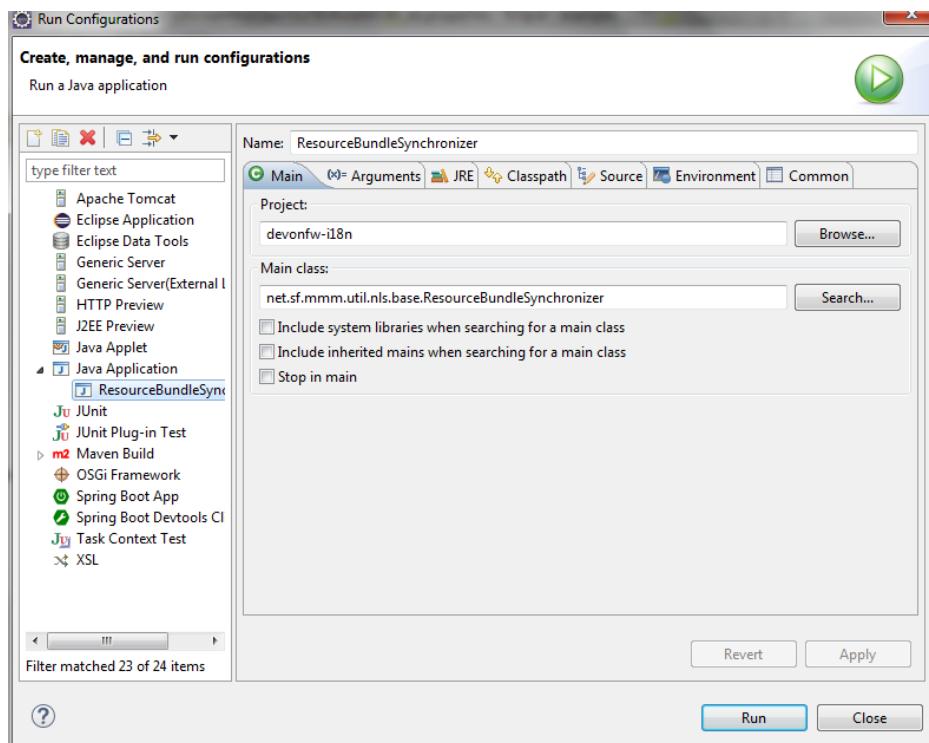
For server-side i18n mmm library is recommended. More details for mmm can be found at <http://m-mm.m.sourceforge.net/apidocs/net/sf/mmm/util/nls/api/package-summary.html> For example , if user wants to use mmm library for internationalization, he has to make an entry in config.properties as

```
i18n.input.name=mmm
```

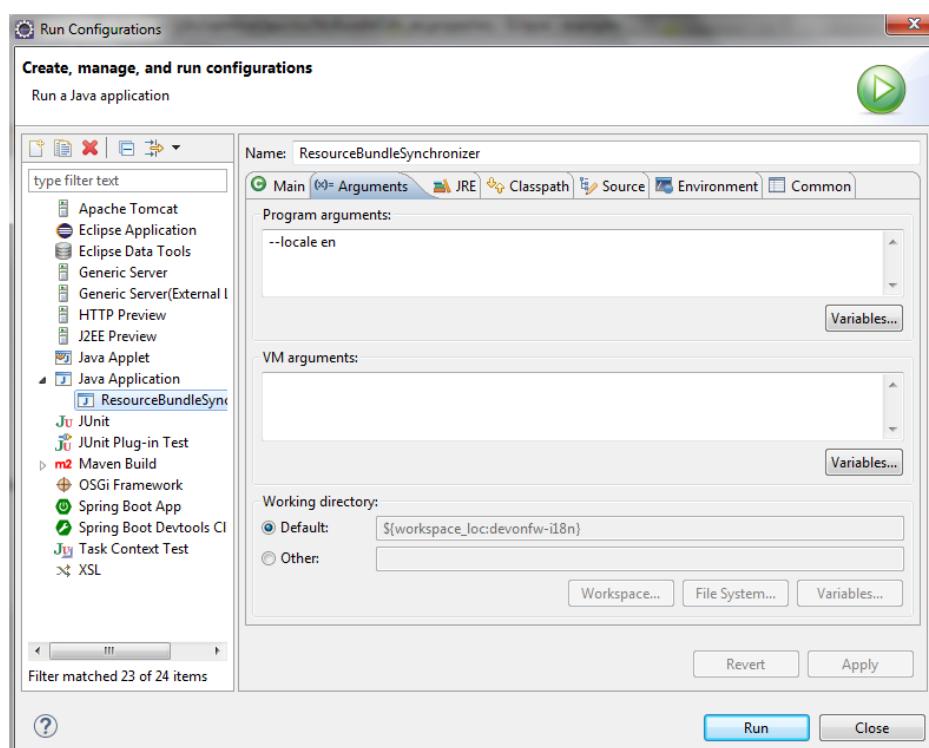
'mmm' is default mechanism for i18n.

Reading property files via MMM implementation:

net.sf.mmm.util.nls.base.ResourceBundleSynchronizer is used to create .properties files for the locales via MMM implementation. Steps to create locales, property files, via MMM: Below are steps to produce locale files at location \src\main\resources\com\capgemini\devonfw\module\i18n\common\api\i18n - Right click on 'i18n' module. - Go to Run As > Run Configurations - Right click on 'Java Application' - Click on 'New' - In the dialog box that is displayed , provide name for the configuration (eg. ResouceBundleSyncronizer) and provide main class name as net.sf.mmm.util.nls.base.ResouceBundleSyncronizer



- Click on Arguments tab besides Main tab.
- Enter program Arguments as "--locale <locale>" eg. "--locale en"



Apply the changes and click 'Run' button. File in config.properties file will have below property:

```
i18n.input.name=mmm
```

To enable i18n functionality in oasp4j based application we need to follow below steps: - Maven clean and build your application - Maven clean build devonfw-i18n with below dependency commented:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
</dependency>
```

- Add following dependency to your oasp4j application. Verify that the *version* matches the last available version of the module:

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-i18n</artifactId>
  <version>2.2.0</version>
</dependency>
```

## Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case, you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command `mvn install`. If the project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

- Comment below statement from *SpringBootApp.java*:

```
@EntityScan(basePackages = { "test.cg.i18nConfigSample" }, basePackageClasses =
{ AdvancedRevisionEntity.class })
```

- Add below statement to *SpringBootApp.java* class:

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.i18n",
"my.other.component" }, basePackageClasses = { AdvancedRevisionEntity.class })
```

Here *my.other.component* refers to any other package which user needs to scan. User should provide *basePackages* from *@EntityScan* annotation. Refer below figure for example:

```
@SpringBootApplication(exclude = { EndpointAutoConfiguration.class })
// @EntityScan(basePackages = { "test.cg.i18nConfigSample" }, basePackageClasses = { AdvancedRevisionEntity.class })
@EnableGlobalMethodSecurity(jsr250Enabled = true)
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.i18n",
"test.cg.i18nConfigSample" }, basePackageClasses = { AdvancedRevisionEntity.class })
public class SpringBootApp {
```

- Add below statement to *ServiceConfig.java* :

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module" })
```

- In *config.properties* set module name which you want to use for i18n- Available modules are "mmm" and "standard". Note: You can create add module as well. Refer to section add own module in i18n.
- Once above changes are done clean build your project in eclipse and launch *SpringBootApp.java*. User can view i18n REST service in available REST webservices (<http://localhost:8081/oasp4j-sample-server/services/rest/>)
- To test i18n REST service, the general format of the service will be as follows:

```
<service root>/locale/<locale indicator>  
eg. localhost:8081/oasp4j-sample-server/services/rest/i18n/locales/en_US
```

[[cookbook-i18n-module\_standard-library (i18n module use com.google.code.gson to convert java property file to json)]] ===== standard library (i18n module use com.google.code.gson to convert java property file to json)

To use standard library from i18n module, user needs to set 'i18n.input.name' property value to 'standard' in config.properties.

```
i18n.input.name=standard
```

This library use com.google.code.gson to convert java property file to json. This data will be returned to user via REST call.

[[cookbook-i18n-module\_customized-library(adding own module in i18n)]] ===== Customized library(Adding own module in I18n)

To add own module in i18n user needs to follow below step:

1. Create new module which will be able to return json data from method call.
2. Add dependency of this module in devonfw-i18n module.
3. In config.properties set i18n.input.name =USER\_MODULE\_NAME
4. In class com.capgemini.devonfw.module.i18n.logic.impl.I18nImpl modify getResourceObject() method add your switch case in it.
5. Clean and build your application and launch SpringBootApp.java. You can view i18n REST service in available REST webservices (<http://localhost:8081/oasp4j-sample-server/services/rest/>)
6. To test i18n REST service, the general format of the service will be as follows:

```
<service root>/locale/<locale indicator>  
eg. localhost:8081/oasp4j-sample-server/services/rest/i18n/locales/en_US
```

## 3.5 Devon-locale module(i18n resource converter)

### 3.5.1 Introduction

Devon-locale is a standalone module; basically it is resource converter for internationalization. Currently, each part of an application - OASp4J, Angular2 etc - use their native resource formats for internationalization. This becomes unwieldy in large systems, especially if part of the texts is shared by the server, client and/or different client apps. Devon-locale solves this problem by using one resource file as input and translating it to one or more target formats. Currently we have java property file as input file, user will mention key value pair in java property file. Current version of devon-locale is converting or translating it to JSON (format used by angular application) and EXTJS (format used by Sencha application) format. So, conceptually this module is a "translator", a custom made program which is able to translate - or transform - to any possible destination format.

[[cookbook-i18n-resource-converter\_devon-locale structure and working ]] == Devon locale structure and working

Devon-locale will convert java properties file to destination format(EXTJS or JSON). We will create IR(Intermediate representation) from java properties file. This IR will be same for all output formats. We will pass this Intermediate representation to various output adapters which will generate respective output files. Currently JsonTargetAdapter and ExtJsTargetAdapter are available. IR is tree like structure created from properties file key value pair.

[[cookbook-i18n-resource-converter\_devon-locale basic usage]] === Devon locale basic usage

- Import source code in eclipse from path <https://github.com/devonfw/devonfw-locale>
- Export as runnable jar.
- Go to command prompt navigate to the path of jar.
- Command to run devon-locale
  - Read from file and write to file

```
java -jar devon-locale.jar -input D:\Test_Dist\Devon-dist_2.1.0_SNAPSHOT\workspaces\i18nWs\TestRepo\devon-locale\src\main\resources\sample.properties -informat java -outformat ANGULAR -output D:\temp.json
```

OR

```
java -jar devon-locale.jar -i D:\Test_Dist\Devon-dist_2.1.0_SNAPSHOT\workspaces\i18nWs\TestRepo\devon-locale\src\main\resources\sample.properties -f java -t ANGULAR -o D:\temp.json
```

- Read from console and write to console

```
java -jar devon-locale.jar -informat java -outformat ANGULAR -input "farewell=Tschüß"
```

OR

```
java -jar devon-locale.jar -f java -t ANGULAR -i "farewell=Tschüß"
```

As you can see in above example there are various input params which are described in below table:

Name	Description	Optional
input, i	File containing source translation	True, default value

Name	Description	Optional
informat, f	Format of the source translation. Possible values: "java"	True, default value "java"
output, o	File with target translation	True, default value"
outformat, t	Format of the target translation Possible values: "angular" or "extjs"	True, default value "angular"

Alternatively we can launch devon-locale from eclipse , we need to run DevonLocale.java as java application.

Sample.properties file used to test above module.

```
object.man.name=John
a.b.c.d.e=f
zz=bb
noFiles=n' y as pas de fichiers
oneFile = y a un fichier
multipleFiles = y a {2} fichiers
pattern = Il {0} sur {1}.
sampleStmt=At 10:16 AM on July 31, 2009, we detected 7 spaceships on the planet Mars. we paid $1000. He
is at position {{0}} \n
greetings = Hallo.
farewell = Tschüß.
inquiry = Wie geht's?
sampleDate=12-10-2017
sampleMultiline= hi \
I am \
Sneha
nextline=new
```

This sample.properties contains

- a. Dates
- b. Multiline input
- c. positional parameters ({0}, {1}, etc)
- d. Special character like üß

## 3.6 The async module

### 3.6.1 Introduction

JAX-RS 2.0 offers a new feature which provides an asynchronous processing at both, the server and the client side.

In the synchronous request/response processing model, client connection is accepted and processed in a single I/O thread by the server. Generally, a pool of I/O threads is available at server side. Therefore, when a request is received at server side, the server dedicates it to one of the threads for further processing. The thread blocks, until the processing is finished and returned.

The idea behind the asynchronous processing model is to separate connection acceptance and request processing operations. Technically, it means to allocate two different threads, one to accept the client connection (acceptor) and the other to handle heavy and time consuming operations (worker) releasing the first one.

[[cookbook-async-module\_the-async ip module]] == The async IP module The above concept is implemented in Devonfw through the *async module*. Therefore, you can include it in your project and will be able to apply the async process of requests in the REST services of your apps in a few steps, avoiding the complex details of the implementation.

[[cookbook-async-module\_adding-the async module dependency to your project]] === Adding the async module dependency to your project To access the functionality of the *async module*, you will need to include its dependency in your project's pom.xml. Verify that the *version* matches the last available version of the module.

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-async</artifactId>
  <version>2.2.0</version>
</dependency>
```

#### Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case, you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command *mvn install*. If the project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-async-module\_injecting-the module]] === Injecting the module

Add the reference to the module in your REST service using the `@Inject` annotation

```
import com.capgemini.devonfw.module.async.common.api.Async;
...
@Service("myRestService")
@Path("/rest")
public class Rest {
  @Inject
  private Async async;
```

```
[ ... ]  
}
```

Remember to add the package of the module to the `@ComponentScan` annotation in the Spring Boot main class.

```
@ComponentScan(basePackages = { "com.capgemini.devonfw.module.async" , "my.other.components.package" })
```

[[cookbook-async-module\_call-the module]] === Call the module Before calling the module, you will need to complete two previous steps:

- Wrap your long process in the `run` method of a class that implements the `AsyncTask`.

```
public class MyAsyncTask implements AsyncTask {  
  
    @Override  
    public Object run() {  
  
        // Here your code for long process  
    }  
  
}
```

- Provide an Async Response. To do so, use the `@Suspended` annotation and add the `AsyncResponse` object in your REST method.

Now, you can call the module using its `execute` method. The request will be bound to the async response and the async task provided.

The call will be like the following:

```
import com.capgemini.devonfw.module.async.common.api.Async;  
import javax.ws.rs.container.AsyncResponse;  
import javax.ws.rs.container.Suspended;  
  
...  
  
@Service("myRestService")  
@Path("/rest")  
public class Rest {  
  
    @Inject  
    private Async async;  
  
    @GET  
    @Path("/asynctask")  
    @Produces(MediaType.TEXT_PLAIN)  
    public void asyncTask(@Suspended final AsyncResponse response) {  
  
        this.async.execute(response, new MyAsyncTask());  
    }  
}
```

That's all. After above simple steps, you will have an async process implemented. Now, let's see some more features of the module.

## Passing parameters

You can also pass the parameters to be used, in the long task process.

In this case, the module call would be:

```
@GET  
@Path("/asynctask/{id}")  
@Produces(MediaType.TEXT_PLAIN)  
public void asyncTask(@Suspended final AsyncResponse response, @PathParam("id") String id) {  
  
    this.async.execute(response, new MyAsyncTask(id));  
}
```

And the wrapper class:

```
public class MyAsyncTask implements AsyncTask {  
  
    private String id;  
  
    public MyAsyncTask(String id) {  
        this.id = id;  
    }  
  
    @Override  
    public Object run() {  
  
        // Here your code for long process with access to 'this.id'  
    }  
}
```

## Module Configuration

Internally, the Async module process can be configured in two main parameters:

- **core pool size:** Sets the ThreadPoolExecutor's core pool size.
- **time out:** The amount of time that the process will wait for the long task, to be finished before return. A timeout of < 0, will cause an immediate return of the process. A timeout of 0, will wait indefinitely.

The default values provided in the module are:

- core pool size: 10.
- time out:
  - milliseconds: 0.
  - status: 503 , service unavailable (available status 400,403,404,500 and 503).
  - response Content: Operation timeout (the time out response message).
  - mediatype: text/plain (you can respond the timeout in json, xml, html, etc. formats).

However, you can edit those values by overriding the configuration properties in your app. To do it, you can use the `application.properties` to add the properties you want to define.

Table 3.1. `application.properties` file

Property	Application Property Name
core pool size	devonfw.async.corePoolSize
time out milliseconds	devonfw.async.timeout.milliseconds
time out status	devonfw.async.timeout.status

time out response content	devonfw.async.timeout.responseContent
time out media type	devonfw.async.timeout.mediatype

As an example, the next could be a valid `application.properties` configuration file, for an application in which, you want an `async` process with a `core pool size` of 20, and a `timeout` of 10 seconds, returning with a status of 500 (internal server error) and a response in `json` format:

```
devonfw.async.corePoolSize=20
devonfw.async.timeout.milliseconds=10000
devonfw.async.timeout.status=500
devonfw.async.timeout.mediatype=application/json
devonfw.async.timeout.responseContent={"response": [{"message": "error", "cause": "time out"}]}
```

## 3.7 The Integration Module

### 3.7.1 Introduction

Within the *Enterprise architecture* the **integration** (or [Enterprise Integration](#)) is the field focused on communication between systems: system interconnection, electronic data interchange, product data exchange, distributed computing environments, etc. So the *integration* defines and provides an infrastructure to allow the communication between different applications or systems in a reliable way. The scope of *Integration* groups a wide range of communication methods, the systems can communicate by file transfer, sharing a data base, by remote procedure invocation or by messaging. The approach of the *Integration Module* implementation is based on the latter, using a solution based on *channels* and *message queues* to allow the exchange of information between applications.

[[cookbook-integration-module\_stack-of technologies]] == Stack of technologies

The *Integration IP Module* is a Java module based on [Spring Integration](#) solution and internally is implemented using the [Spring integration Java dsl reference](#). In the core of all those technologies the implementation relies on the [Java Message Service \(JMS\)](#) as the messaging standard to create, send, receive and read messages.

- **Java Message Service:** JMS allows applications to exchange messages using reliable and loosely coupled communication. To achieve this, the communication is done through message queues so the different applications (message server, and message clients) don't know the actual addresses of each other, so they work in an emitter/subscriber manner. To manage those message queues, that are "out" of the applications but should be accessible, the solution provided by Devon relies on a message broker that will be in charge of messages infrastructure.
- **Apache Active MQ:** The *channels*, *queues* and *messages* that the solution is based on, need an external infrastructure to be supported. The Devon implementation relies on [Apache Active MQ](#) as the message broker to manage the messages and queues that the different applications will use as communication channel. Active MQ is an open source infrastructure that is one of the most popular messaging servers and provides fully support for *JMS*.
- **Spring Integration** extends the Spring framework to support the Enterprise Integration (system interconnection, electronic data interchange, product data exchange and distributed computing environments). Spring Integration enables lightweight messaging within Spring-based applications providing a simple model for building enterprise integration solutions while maintaining the separation of concerns. Among its features Spring Integration provides JMS support through Channel Adapters for receiving JMS messages.
- The **Java DSL extension for Spring Integration** provides a set of convenient Builders and a fluent API to configure Spring Integration message flows from Spring configuration classes. It allows the creation of channel/queues and the message flow in applications start up time, so users can avoid the standard Spring Integration XML configuration (although it can also be used along side the DSL definitions).

[[cookbook-integration-module\_installing-apache active mq]] == Installing Apache Active MQ

[[cookbook-integration-module\_how-to get and install the message broker]] === How to get and install the message broker

If you want to be ready to a quick test of the module you will need an *Active MQ* broker. To do so you can download the *Active MQ* message broker from the [official site](#). After the download is over, extract

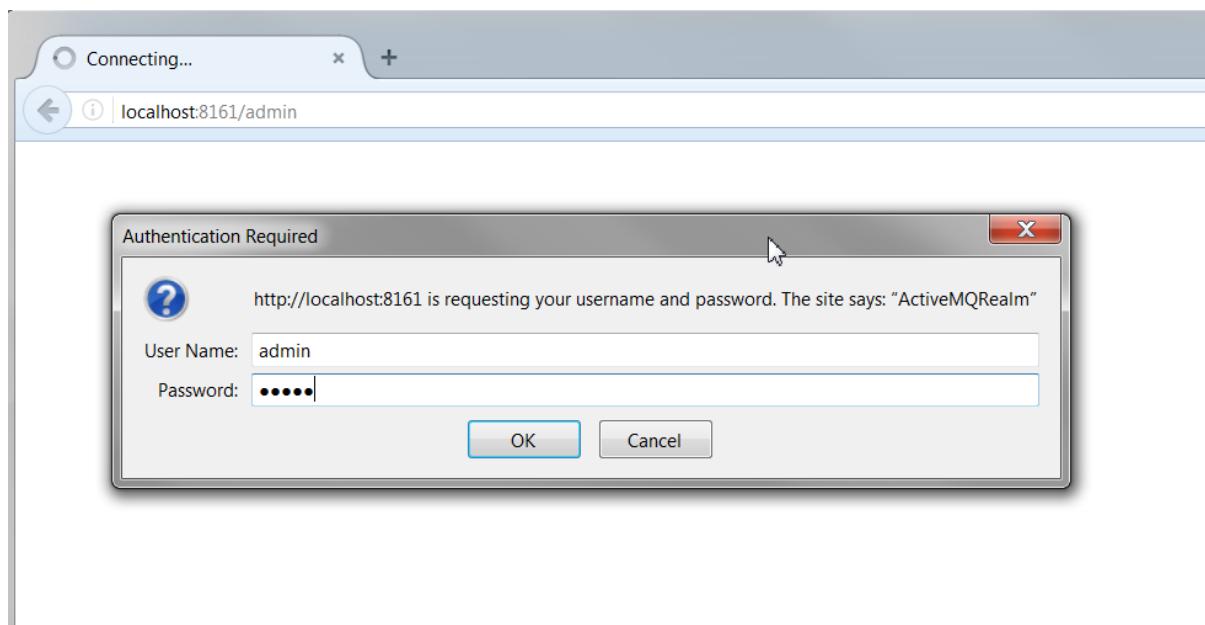
the zip folder, open a command line window and go to the *bin* folder inside the just created *apache-activemq-{version}* folder.

```
your\location\apache-activemq-5.14.3>cd bin
```

And start the server from the bin folder

```
your\location\apache-activemq-5.14.3\bin>activemq start
```

Now open a browser and access to url <http://localhost:8161/admin>, fill the access login with admin for both *user* and *password*.



After logging in, you will have access to all the infrastructure of the message server

## Note

We are installing the Active MQ server in our local machine only for test or example purposes. If you want further configuration details please visit the [official documentation](#) or ask your IT department.

[[cookbook-integration-module\_using-apache active mq with docker]] == Using Apache Active MQ with Docker

Active MQ can also be run with Docker, you will need to:

- Create a folder on your machine for a shared volume with the container (c:/Users/docker/activemq on the example below)
- Create an activemq.xml configuration file (c:/Users/docker/activemq/conf/activemq.xml on the example below)
- Run the following command

```
docker run --rm -p 61616:61616 -p 8161:8161 -v c:/Users/docker/activemq/conf:/etc/activemq/conf -v c:/Users/docker/activemq/data:/var/activemq/data rmohr/activemq:5.14.3-alpine
```

For the activemq.xml file, you can use the following content (extracted from sample folder on activemq distribution)

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.apache.org/schema/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://activemq.apache.org/schema/core http://activemq.apache.org/schema/core/activemq-core.xsd
```

```
http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

<broker xmlns="http://activemq.apache.org/schema/core" useJmx="false">

  <persistenceFactory>
    <journalPersistenceAdapterFactory journalLogFiles="5" dataDirectory="..../data"/>
  </persistenceFactory>

  <transportConnectors>
    <transportConnector uri="tcp://localhost:61616"/>
    <transportConnector uri="stomp://localhost:61613"/>
  </transportConnectors>

</broker>

</beans>
```

[[cookbook-integration-module\_module-connection configuration]] === Module connection configuration

The *Integration module* provides a default connection configuration for Active MQ broker through the following properties

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

If you have changed the *Active MQ* configuration remember to overwrite the affected properties in the application.properties of your project.

[[cookbook-integration-module\_ip-integration module details]] == IP Integration module details

[[cookbook-integration-module\_adding-the module to a project]] === Adding the module to a project  
To access the functionality of the *Integration module*, you will need to include its dependency in your project's pom.xml. Verify that the *version* matches the last available version of the module.

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-integration</artifactId>
  <version>2.2.0</version>
</dependency>
```

## Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case, you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command *mvn install*. If the project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-integration-module\_injecting-the module]] === Injecting the module

After adding the dependency, in order to start using the module inject it using the *@Inject* annotation

```
import com.capgemini.devonfw.module.integration.common.api.Integration;
```

```
...
```

```
@Inject
private Integration integration;
```

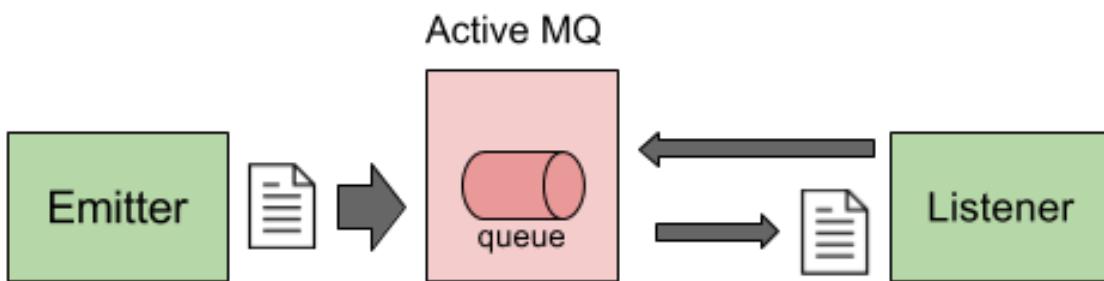
We will see the detailed usage of the module in the upcoming sections.

## Default channels

Based on the mentioned stack of technologies, the Devonfw *Integration module* provides three communication channels pre-configured and ready to be used out-of-the-box. The user will only need to *enable* it through the module configuration.

[[cookbook-integration-module\_simple-message channel]] ===== Simple message channel

This is the most basic communication channel. In this case in one side is an application (*emitter*) that sends messages to a specific queue in the message broker. In the other side a second application (*listener*) is *subscribed* to that channel, which means that polls the message broker in a defined interval of time to ask for new messages in that particular queue.



The *listener* application doesn't provide a response, only consumes the messages.

To configure your application to use this default channel you only need to edit the `application.properties` of your Spring project adding the property `devonfw.integration.one-direction.emitter` or `devonfw.integration.one-direction.listener`.

For emitter applications set the `one-direction.emitter` property to *true*:

```
devonfw.integration.one-direction.emitter=true
```

If your application acts as listener set the property `one-direction.listener` to *true*:

```
devonfw.integration.one-direction.listener=true
```

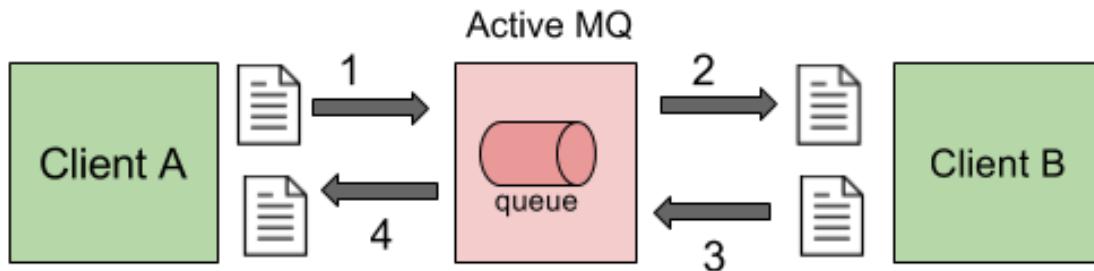
Doing this, when running your app the related *Beans* will be loaded automatically and the communication channel and its related queue will be also created.

We will see more details of the simple message channel configuration further.

[[cookbook-integration-module\_request-reply-channel ]] ===== Request-Reply channel

In this second approach the message flow is completed in two directions. In this case, instead of talking about an *emitter* and *listener* systems, we should rather talk about a *request/replay* channel. There will

exist a communication between two clients, in which the first one will send a message and wait for a response from the second one. So both sides are *emitters* and *listeners*.



To configure your application to use this default channel, as we explained in the previous section, you only need to edit the `application.properties` of your Spring project adding in this case the property `devonfw.integration.request-reply.emitter` or `devonfw.integration.request-reply.listener`.

For emitter/listener applications set the `request-reply.emitter` property to `true`:

```
devonfw.integration.request-reply.emitter=true
```

If your application acts as listener/emitter set the property `request-reply.listener` to `true`:

```
devonfw.integration.request-reply.listener=true
```

Doing this, same as in the previous case, when running your app the related *Beans* will be loaded automatically and the communication channel and its related queue will be also created.

We also will see more details of the simple message channel configuration further.

[[cookbook-integration-module\_request-reply-asynchronous channel]] ===== Request-Reply asynchronous channel

For the cases where the previous `request/reply` communication has to be `asynchronous` the module provides a default `async` communication channel.

To configure your application to use this asynchronous channel, as in the previous cases, you only need to enable the corresponding properties into the `application.properties` file of your project.

For emitter/listener applications set the `request-reply-async.emitter` property to `true`:

```
devonfw.integration.request-reply-async.emitter=true
```

Otherwise, if your application is the listener/emitter, set the property `request-reply.listener` to `true`:

```
devonfw.integration.request-reply-async.listener=true
```

We will show the complete configuration of this default channel in upcoming sections.

[[cookbook-integration-module\_usage-of the default channels]] === Usage of the default channels

[[cookbook-integration-module\_how-to use the default simple channel]] ===== How to use the default simple channel

As we previously mentioned the *Integration module* provides a simple communication channel where in one side one *emitter* application will send a message and in the other side other *listener* application will receive and read it.

To achieve that in our applications we only need to configure the corresponding properties to create the channel and its related queue.

### Default configuration

The default configuration properties for this channel, provided by default with the *Integration module*, are:

```
devonfw.integration.one-direction.emitter=false
devonfw.integration.one-direction.listener=false
devonfw.integration.one-direction.channelname=1d.Channel
devonfw.integration.one-direction.queuename=1d.queue
devonfw.integration.one-direction.poller.rate=5000
```

- *emitter*: if your app is going to send messages through this channel to the related queue.
- *listener* if your app is going to subscribe to the channel to read the messages of the queue.
- *channelname*: the name for the channel.
- *queuename*: the name for the channel queue.
- *poller.rate*: in case of listener applications this is the interval to poll the message broker for new messages.

If you want to customize these properties you can overwrite them in the `application.properties` of your project.

[[cookbook-integration-module\_emitter-application configuration]] ===== Emitter application configuration

As we already mentioned the *Emitter* applications must enable the *emitter* property so you must add to the `application.properties` file of your project the property

```
devonfw.integration.one-direction.emitter=true
```

Optionally, you can edit the name for the channel and for the queue using the `devonfw.integration.one-direction.channelname` and `devonfw.integration.one-direction.queuename` properties.

[[cookbook-integration-module\_emitter-application example]] ===== Emitter application example

After you have added the [module dependency](#) you can start using the module [injecting it](#) in your app. Lets see how to send a simple message through that default *simple channel*.

#### Note

In order to make the example run properly remember that we will need an [Active MQ](#) instance running to provide support to the *channels* and *queues*.

In our *sender* application we only need to call the *send* method of the *Integration* object and provide a message content

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyEmitterApp{

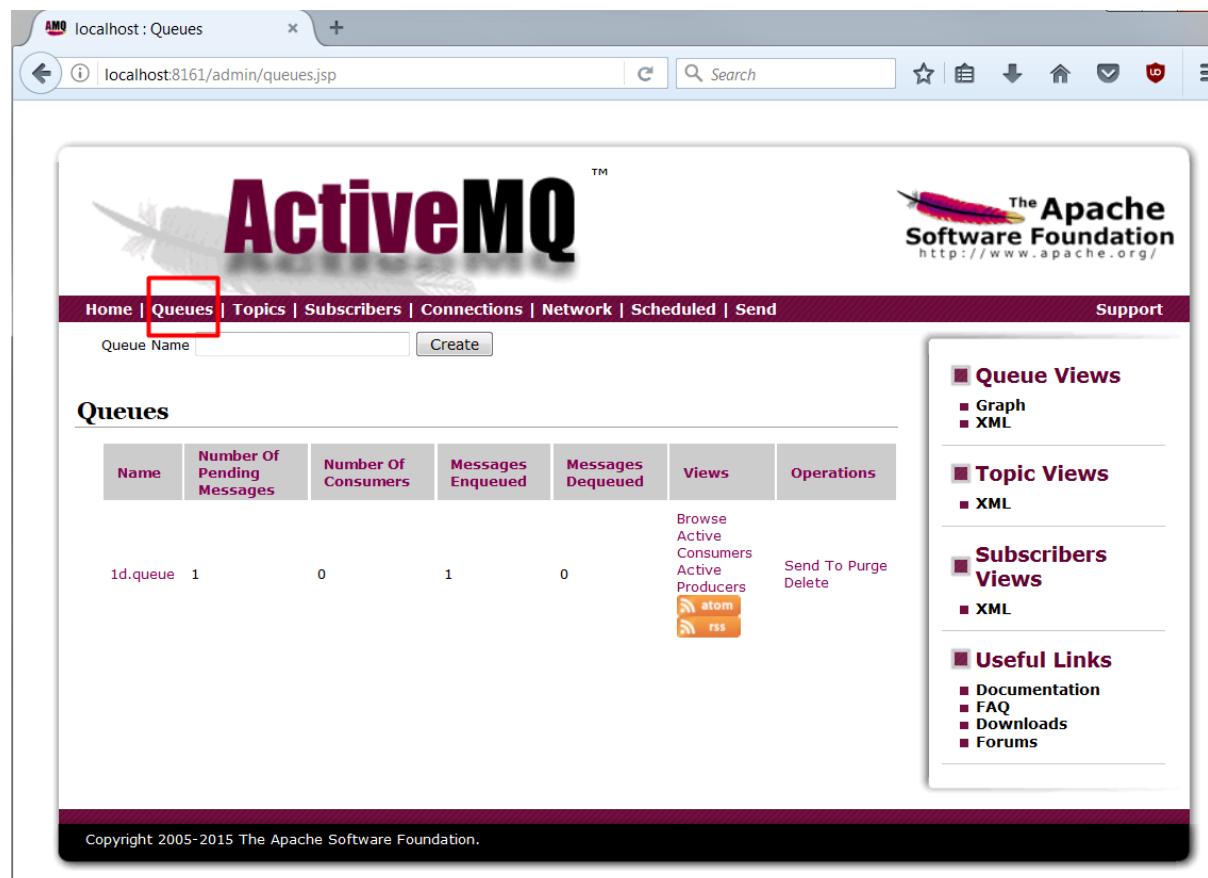
    @Inject
    private Integration integration;

    public void sendSimpleMessage(){

        this.integration.send("hello world");
    }

}
```

Running the application will result into a message sent to the *Integration module* default *simple channel* with name `1d.Channel` and to the queue `1d.queue` (or the names you provided through configuration properties). So if now we go to the Active MQ web client we will see in the *Queues* section that we have a new queue created with one message as *pending messages*, no *consumers* (as we still don't have any subscriber to this *channel/queue*) and no *dequeued messages*.



Queues

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
1d.queue	1	0	1	0		<a href="#">Browse</a> <a href="#">Active Consumers</a> <a href="#">Active Producers</a> <a href="#">Send To</a> <a href="#">Purge</a> <a href="#">Delete</a>

Operations

Views

Operations

Queues

Topics

Subscribers

Connections

Network

Scheduled

Send

Support

Queue Views

Graph

XML

Topic Views

XML

Subscribers Views

XML

Useful Links

Documentation

FAQ

Downloads

Forums

Copyright 2005-2015 The Apache Software Foundation.

Clicking on the queue name shows us the pending messages details

**Browse id.queue**

Message ID	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:LES002610-51097-1489050172329-1:1:1:1		Persistent	4	false		2017-03-09 10:02:52:860 CET		<a href="#">Delete</a>

[View Consumers](#)

And clicking again on the *message ID* takes us to the *message view* where we can see more details like the message content

Headers	Properties
Message ID: ID:LES002610-51097-1489050172329-1:1:1:1	timestamp: 1489050172174
Destination: queue://id.queue	
Correlation ID:	
Group:	
Sequence: 0	
Expiration: 0	
Persistence: Persistent	
Priority: 4	
Redelivered: false	
Reply To:	
Timestamp: 2017-03-09 10:02:52:860 CET	
Type:	

**Message Actions**

[Delete](#)  
[Copy](#)  
[Move](#) -- Please select --

**Message Details**

Hello world

With this we have finished the *out flow* for the *Integration module* default *simple channel*. Lets see now how to read that message we have sent using a different application.

[[cookbook-integration-module\_listener-application configuration]] ===== Listener application configuration

For *Listener* applications you must enable the channel through the corresponding property in the *application.properties* file of your project.

```
devonfw.integration.one-direction.listener=true
```

In case of *listener* applications you can also configure the interval of time to make the requests to the message broker for new messages. To do so you can add the property *devonfw.integration.one-*

direction.poller.rate to your application.properties file and provide a milliseconds amount as property value. If you don't overwrite this property its default value is 5000 (5 seconds).

As in the *emitter* case, you can edit the name for both the *channel* and the *queue* (devonfw.integration.one-direction.channelname and devonfw.integration.one-direction.queuename properties) but have into account that **these names must match** between the *emitter* and the *listener* applications in order to perform the communication.

[[cookbook-integration-module\_listener-application example]] ===== Listener application example

As in the case of *emitter* application you have to add the [module dependency](#) and [inject](#) the module. Once that is done we can subscribe our application to the *channel/queue* to start receiving messages from the *Integration module* default simple channel.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyListenerApp{

    @Inject
    private Integration integration;

    @Inject
    private MessageHandler messageHandler;

    public void readSimpleMessage(){

        this.integration.subscribe(this.messageHandler);
    }
}
```

In this case we provide to the `subscribe` method a *Message Handler* to manage what we want to do with each message. For the example we have implemented a basic message handler. To create your own *Message Handler* you only need to create a class, annotate it with `@Handler` and implement the `MessageHandler` interface. Lets see our `SimpleMessageHandler`

```
@Component
public class SimpleMessageHandler implements MessageHandler {

    public void handleMessage(Message<?> message) throws MessagingException {

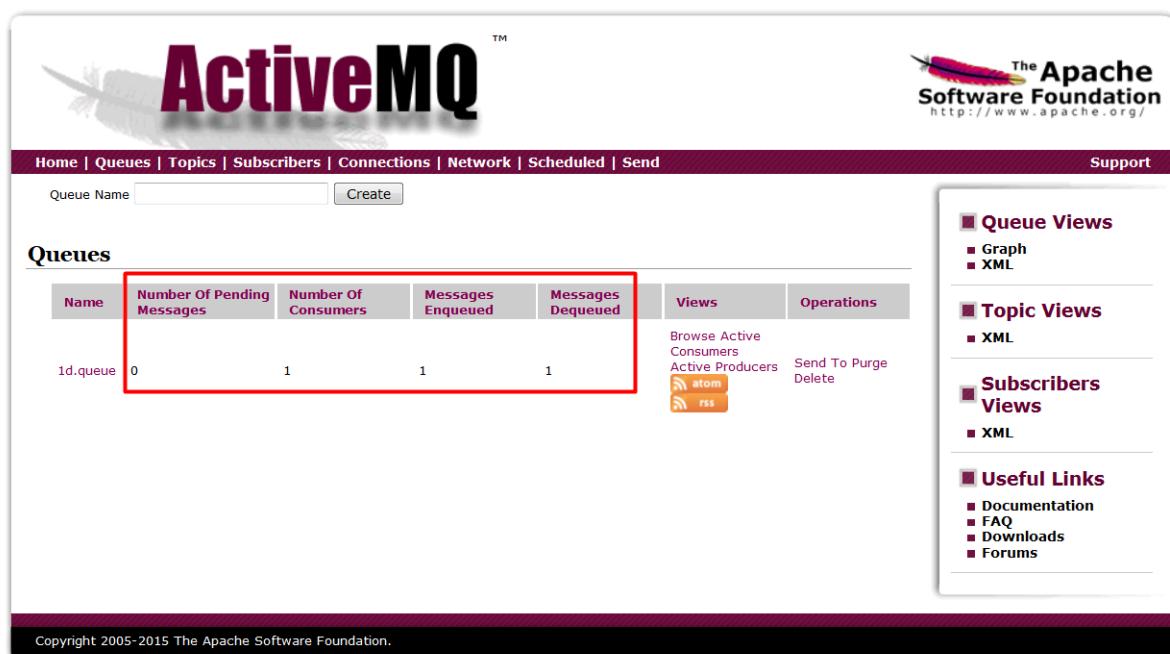
        System.out.println("*****");
        System.out.println("MESSAGE IS: " + message.getPayload());
        System.out.println("*****");

    }
}
```

As you can see we are only showing, through console, the message content which we access through the `getPayload()` method. Now running the application we get the output

```
*****
MESSAGE IS: Hello world
*****
```

And going back to the Active MQ web client we can see the changes in the `Queues` section



The screenshot shows the ActiveMQ web interface. At the top, there is a navigation bar with links: Home, Queues, Topics, Subscribers, Connections, Network, Scheduled, Send, and Support. Below the navigation bar is a search bar with 'Queue Name' and a 'Create' button. The main area is titled 'Queues' and shows a table with one row for '1d.queue'. The table columns are: Name, Number Of Pending Messages, Number Of Consumers, Messages Enqueued, Messages Dequeued, Views, and Operations. The 'Pending Messages' column for '1d.queue' is highlighted with a red box and contains the value '0'. The 'Consumers' column contains '1', 'Messages Enqueued' is '1', and 'Messages Dequeued' is '1'. To the right of the table are 'Views' (Browse Active Consumers, Active Producers) and 'Operations' (Send To Purge, Delete) buttons. Below the table are links for atom and rss feeds. On the right side of the interface, there is a sidebar with sections: Queue Views (Graph, XML), Topic Views (XML), Subscribers Views (XML), and Useful Links (Documentation, FAQ, Downloads, Forums). At the bottom of the interface, there is a copyright notice: 'Copyright 2005-2015 The Apache Software Foundation.'

The first you should note is that now the *Number of Consumers* is *1* as we have subscribed an application to the channel. Then the *Pending Messages* has changed to *0* and the *Messages Dequeued* has increased to *1*.

At this point we have finished the example for the *in flow* of the *Integration module* default *simple channel*. Now you have the whole picture of how a simple integration channel works with Devonfw *Integration module* and *Active MQ* server.

[[cookbook-integration-module\_how-to use the default request-reply channel]] ===== How to use the default request-reply channel

With the *Integration module* a ready to be used *request-reply* channel is provided by default. This channel will allow us to communicate systems sending and receiving messages in both sides. A first application will send a message and wait for a response, while a second application will receive the message sent by the first one and send back the response that the other app is waiting for.

To achieve that in our applications we only need to configure the corresponding properties to create the channel and its related queue.

[[cookbook-integration-module\_default-configuration for request-reply channel]] ===== Default configuration for request-reply channel

The default configuration properties for this channel, provided by default with the *Integration module*, are:

```
devonfw.integration.request-reply.emitter=false
devonfw.integration.request-reply.listener=false
devonfw.integration.request-reply.channelname=rr.Channel
devonfw.integration.request-reply.queuename=rr.queue
devonfw.integration.request-reply.receiveTimeout=5000
```

- *emitter*: if your app is going to send and then receive messages through this channel.
- *listener*: if your app is going to receive and then send back messages using this channel.

- *channelname*: the name for the channel.
- *queuename*: the name for the channel queue.
- *receivetimeout*: in case of send + receive applications this is the maximum amount of milliseconds to receive a response from "the other side" of the channel. If this time is exceeded a timeout *Exception* will be thrown.

If you want to customize these properties you can overwrite them in the `application.properties` of your project, as we are going to see below.

`[[cookbook-integration-module_sender-receiver-application configuration]] ===== Sender-Receiver application configuration`

To enable the sending of messages through this channel you must set the `request-reply.emitter` property to `true` in the `application.properties` of our project.

```
devonfw.integration.request-reply.emitter=true
```

Optionally, you can edit the name for the channel and for the queue using the `devonfw.integration.request-reply.channelname` and `devonfw.integration.request-reply.queuename` properties. As we just mentioned, the timeout for the response can be edited adding the `devonfw.integration.request-reply.receivetimeout` property to our properties file and providing a milliseconds value. By default the timeout is 5000 (5 seconds).

`[[cookbook-integration-module_sender-receiver application example]] ===== Sender-Receiver application example`

After you have added the [module dependency](#) you need to [inject it](#). Lets see how to send and receive a simple message through that default *request-reply* channel.

### Note

In order to make the example run properly remember that we will need an [Active MQ](#) instance running to provide support to the *channels* and *queues*.

In our *sender-receiver* application we only need to call the `sendAndReceive` method of the *integration* object and provide a message content

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyFirstApp{

    @Inject
    private Integration integration;

    public void myMethod(){

        String response = this.integration.sendAndReceive("Hello");
        System.out.println("Response:" + response);
    }
}
```

If now we run the application we would get a *timeout exception* as there is no one ready to provide a response within the defined timeout limit (5 seconds). So first, lets prepare our *other-side* application.

## [[cookbook-integration-module\_receiver-sender-application configuration]] ===== Receiver-Sender application configuration

In this application we need to enable the `request-reply.listener` property so, in the `applications.property` file of our project, we must set to `true` that property.

```
devonfw.integration.request-reply.listener=true
```

You can also edit the name for both the *channel* and the *queue* (`devonfw.integration.request-reply.channelname` and `devonfw.integration.request-reply.queuename` properties). But, as mentioned in previous section, have into account that **these names must match** between the *sender-receiver* and the *receiver-sender* applications, in order to perform the communication.

## [[cookbook-integration-module\_receiver-sender-application example]] ===== Receiver-Sender application example

As in the case of *sender-receiver* application, you have to add the [module dependency](#) and [inject](#) the module. Once that is done we can subscribe our application to the *channel/queue* to start receiving messages and sending responses from/to the *Integration module* default request-reply channel.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MySecondApp{

    @Inject
    private Integration integration;

    @Inject
    private IntegrationHandler simpleHandler;

    public void myMethod(){

        this.integration.subscribeAndReply(this.simpleHandler);
    }
}
```

In this case we provide to the `subscribeAndReply` method an *Integration Handler* to manage the responses to each message. For the example we have implemented a basic *Integration handler*, to create your own one you only need to create a class and implement the `IntegrationHandler` interface. Lets see our `SimpleMessageHandler`

```
public class SimpleIntegrationHandler implements IntegrationHandler {

    @Override
    public Object handleMessage(Message<?> m) {

        System.out.println("*****");
        System.out.println("MESSAGE IS: " + m.getPayload());
        System.out.println("*****");

        return m.getPayload().toString().concat(" World");
    }
}
```

As you can see we are simply printing the original message received, using the `getPayload()` method, and then replying adding to it "World".

At this point we can run that second application and see what happens through the Active MQ web client.

The screenshot shows the ActiveMQ web interface. At the top, there is a navigation bar with links: Home, Queues, Topics, Subscribers, Connections, Network, Scheduled, and Send. On the right, there is a 'Support' link. Below the navigation bar, there is a search bar labeled 'Queue Name' and a 'Create' button. The main content area is titled 'Queues' and shows a table with one row for the queue 'rr.queue'. The table columns are: Name, Number Of Pending Messages, Number Of Consumers, Messages Enqueued, Messages Dequeued, Views, and Operations. The 'rr.queue' row has values: 0, 1, 0, 0, and a set of icons for 'Views' and 'Operations'. The 'Operations' icons include 'Browse Active Consumers', 'Active Producers', 'Send To Purge', and 'Delete'. To the right of the table, there is a sidebar with sections for Queue Views (Graph, XML), Topic Views (XML), Subscribers Views (XML), and Useful Links (Documentation, FAQ, Downloads, Forums). At the bottom of the interface, there is a copyright notice: 'Copyright 2005-2015 The Apache Software Foundation.'

The above image shows that the channel and queue for our request-reply channel have been created automatically and in the *Number of Consumers* you can see that 1 that refers to our application.

Now we can run the first application, as at this point we already have the second application ready to reply to the first one requests.

The output in the second application is as expected

```
*****
MESSAGE IS: Hello
*****
```

While the output in the first app is

```
Response:Hello World
```

If we check out again the Active MQ web client we can see that we still have one consumer (the second application) but now we have also one *Message Enqueued* and one *Message Dequeued*.

Queues

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
rr.queue	0	1	1	1	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a> <a href="#">Send To Purge</a> <a href="#">Delete</a>	<a href="#">Atom</a> <a href="#">RSS</a>

Copyright 2005-2015 The Apache Software Foundation.

We have finished the demonstration for the default Request-Reply channel provided by the *Integration module*. Now we are going to see how to achieve the same but in an asynchronous way using the third default channel provided by the module: the *request-reply-async* channel.

[[cookbook-integration-module\_how-to use the default asynchronous request-reply channel]] ===== How to use the default asynchronous request-reply channel

The usage of this default channel, provided also by default within the *Integration module*, is the same than for previous channels, and specially regarding the default *request-reply* channel explained in the previous section. Anyway let's briefly show the basics about how to configure and use the asynchronous channel.

[[cookbook-integration-module\_default-properties for asynchronous channel]] ===== Default properties for asynchronous channel

```
devonfw.integration.request-reply-async.emitter=false
devonfw.integration.request-reply-async.listener=false
devonfw.integration.request-reply-async.channelname=async.Channel
devonfw.integration.request-reply-async.queueusername=async.queue
devonfw.integration.request-reply-async.receiveTimeout=5000
```

The properties are the same as in the simple *request-reply* channel.

The application that is going to trigger the communication flow, sending a first message, must enable the *request-reply-async.emitter* property, setting *true* as value.

```
devonfw.integration.request-reply-async.emitter=true
```

In the configuration of this application we can also define the timeout for the response. If exceeded, the process will be stopped and a *timeout exception* will be thrown. This can be controlled with the property *devonfw.integration.request-reply-async.receiveTimeout*

In the other side, the application that is subscribed to the channel and is going to receive the messages and reply to them, must have the property *request-reply-async.listener* defined as *true*.

```
devonfw.integration.request-reply-async.listener=true
```

[[cookbook-integration-module\_sender-receiver-async example]] ===== Sender-Receiver async example

After you have added the [module dependency](#) you need to [inject it](#). Lets see how to send and receive asynchronously a message through that default *request-reply-async* channel.

### Note

In order to make the example run properly remember that we will need an [Active MQ](#) instance running to provide support to the *channels* and *queues*.

In our *sender-receiver* application we only need to call the *sendAndReceiveAsync* method of the *integration* object and provide a message content. As we are creating an asynchronous process we will use the Java [Future](#) to handle the response. We will not complicate the example with too many details of Future's use so the code will look like the following

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyFirstApp{

    @Inject
    private Integration integration;

    public void myMethod() throws InterruptedException, ExecutionException{

        Future<String> response = this.integration.sendAndReceiveAsync("Hello");
        System.out.println("Message sent.");
        while (!response.isDone()) {
            // things that you can do in parallel while waiting for the response
            System.out.println("Waiting...");
        }

        System.out.println("ASYNC RESPONSE: " + response.get());
    }
}
```

If now we run the application we would get a *timeout exception* as there is no one ready to provide a response within the defined timeout limit (5 seconds). So first, lets prepare our *other-side* application.

[[cookbook-integration-module\_receiver-sender-async example]] ===== Receiver-Sender async example

As in the previous application, you have to add the [module dependency](#) and [inject](#) the module. Once that is done we can subscribe our application to the *channel/queue* (with the *subscribeAsync* method) to start receiving asynchronously messages and sending responses from/to the *Integration module* default *request-reply-async* channel.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MySecondApp{

    @Inject
    private Integration integration;

    @Inject
    private IntegrationHandler longHandler;
```

```
public void myMethod() {
    this.integration.subscribeAsync(this.longHandler);
}
```

In this case we provide to the `subscribeAsync` method an *Integration Handler* to manage the responses to each message. For the example we have implemented a very simple Integration handler that blocks the process during 3 seconds to simulate a long process. To create your own *Integration Handler* you only need to create a class, annotate it with `@Component` and implement the `IntegrationHandler` interface. Lets see our `LongIntegrationHandler`.

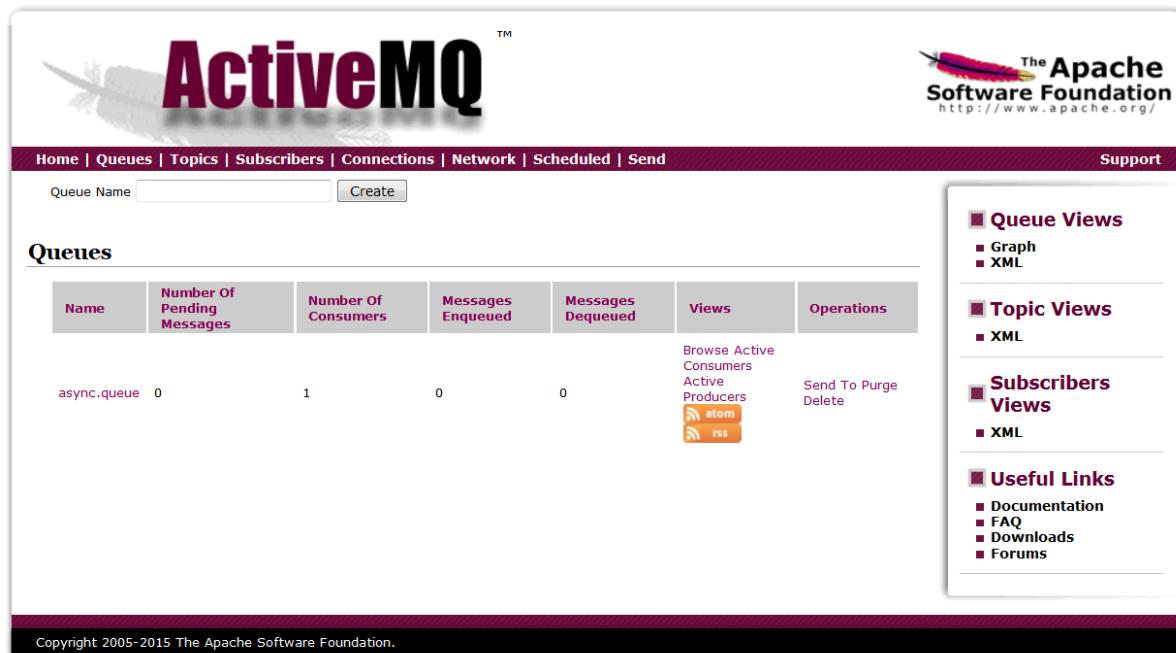
```
@Component
public class LongIntegrationHandler implements IntegrationHandler {

    @Override
    public Object handleMessage(Message<?> m) {

        System.out.println("*****");
        System.out.println("MESSAGE IS: " + m.getPayload());
        System.out.println("*****");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return m.getPayload().toString().concat(" World");
    }
}
```

As you can see we are simply printing the original message received, using the `getPayload()` method, and after the delay of 3 seconds, it returns a reply adding "World" to the original message.

Now we can run that second application, the channel and its `async.queue` will be automatically created in the Active MQ broker and the new consumer (our second app) will be subscribed to that channel.



The screenshot shows the ActiveMQ Administration Console. At the top, there's a navigation bar with links: Home, Queues, Topics, Subscribers, Connections, Network, Scheduled, and Send. On the right, there's a 'Support' link and the Apache Software Foundation logo. The main content area is titled 'Queues'. It contains a table with the following data:

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
async.queue	0	1	0	0	<a href="#">Browse Active Consumers</a> <a href="#">Active Producers</a> <a href="#">atom</a> <a href="#">rss</a>	<a href="#">Send To Purge</a> <a href="#">Delete</a>

On the right side of the page, there's a sidebar with several sections:

- Queue Views**
  - Graph
  - XML
- Topic Views**
  - XML
- Subscribers Views**
  - XML
- Useful Links**
  - Documentation
  - FAQ
  - Downloads
  - Forums

At the bottom of the page, there's a copyright notice: 'Copyright 2005-2015 The Apache Software Foundation.'

If now we run the first application the output is

```
Message Sent.  
[...]  
Waiting...  
Waiting...  
Waiting...  
ASYNC RESPONSE: Hello World
```

With this we have completed the example about the default asynchronous channel for the request-reply flow of the Devonfw *Integration module*.

Also here ends the content about the pre-configured part of the module. Next we will see how to create, programmatically, new channels and queues.

[[cookbook-integration-module\_creating-new channels]] ===== Creating new channels

The Devonfw *Integration module* provides the option of creating new channels programmatically. The user can generate new channels and send and receive messages defining every step in Java code, without the necessity of configure anything.

[[cookbook-integration-module\_types-of channels that can be created]] ===== Types of channels that can be created

The types are the same than the default channels. The user will be able to create

- simple channels: one app sends a message, other app receives the message.
- request-reply channels: a first app sends a message, a second app receives the message and sends a response, the first app receives the response.
- asynchronous request-reply channels: Same as the previous channel but with asynchronous behaviour.

[[cookbook-integration-module\_creating-and using a new simple channel]] ===== Creating and using a new simple channel

After you have added the [module dependency](#) and [injected it](#) you only need to call the `createChannel` method and provide a *name* for the channel and a *name* for the related queue.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;  
  
public class MySenderApp{  
  
    @Inject  
    private Integration integration;  
  
    public void sendSimpleMessage(){  
  
        IntegrationChannel myChannel = this.integration.createChannel("my-channel", "my-queue");  
        Boolean sent = myChannel.send("Hello");  
        if (sent) System.out.println("message successfully sent");  
    }  
}
```

Then, in the *listener* application, after adding the [module dependency](#) and the [injection](#) of it, we only need to use the `subscribeTo` method and provide the name for the channel and the queue (that **must match** the provided names in the first application) and the *Message Handler* to manage the received messages. For more details about the *Message Handler* check out the [listener application example](#) section.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyListenerApp{

    @Inject
    private Integration integration;

    @Inject
    private MessageHandler messageHandler;

    public void readSimpleMessage(){

        this.integration.subscribeTo("my-channel", "my-queue", this.messageHandler);
    }
}
```

By default, the interval for polling the channel is 5000 (5 seconds) and can be changed through property `devonfw.integration.default.poller.rate` in `application.properties` file. In addition, you can define that value when creating the channel passing the milliseconds timeout as a parameter

```
this.integration.subscribeTo("my-channel", "my-queue", this.messageHandler, 10000);
```

[[cookbook-integration-module\_creating-and using a new request-reply channel]] ===== Creating and using a new request-reply channel

In the app that is going to start the flow, after adding the [module dependency](#) and [injected it](#), you only need to call the `createRequestReplyChannel` method and provide a *name* for the channel, a *name* for the related queue and, this part is slightly different from the rest of module implementation, you need to provide a *Message Handler* that will manage the received response, as we do with listener applications.

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MyFirstApp{

    @Inject
    private Integration integration;

    @Inject
    private MessageHandler messageHandler;

    public void startCommunication(){

        IntegrationChannel myChannel = this.integration.createRequestReplyChannel("my-channel", "my-
queue", this.messageHandler);
        Boolean sent = myChannel.send("Hello");
        if (sent) System.out.println("message successfully sent");
    }
}
```

The *MessageHandler* provided in the example above is exactly the same than explained previously in this *Integration module* chapter ([see here](#)), except that, as this is a custom implementation, we don't need to annotate the class with `@Handler`.

With this code we will create the *channel/queue* infrastructure, send the message and provide a handler for the response. Now we need to define the second side of the flow to receive the message and provide a reply.

The timeout for the response can be configured through property `devonfw.integration.default.receivetimeout` in `application.properties` file, by default is set to 5000 (5 seconds).

You can also configure it when creating the channel passing the *timeout* as a parameter

```
IntegrationChannel myChannel = this.integration.createRequestReplyChannel("my-channel", "my-queue", this.messageHandler, 10000);
```

In a second application, after adding the [module dependency](#) and the [injection](#) of it, we only need to use the `subscribeAndReplyTo` method and provide the name for the channel and the queue (that **must match** the names provided in the first app) and the *Integration Handler* to manage the received messages. The implementation is the same as the one described in the [receiver application example](#) section. So our sample code will look like

```
import com.capgemini.devonfw.module.integration.common.api.Integration;

public class MySecondApp{

    @Inject
    private Integration integration;

    @Inject
    private IntegrationHandler myIntegrationHandler;

    public void startCommunication(){

        this.integration.subscribeAndReplyTo("my-channel", "my-queue", this.myIntegrationHandler);
    }
}
```

### Note

Remember that if you run the first app before the listener app is running you will probably get a *timeout exception*.

[[cookbook-integration-module\_creating-and using a new asynchronous request-reply channel]] =====  
Creating and using a new asynchronous request-reply channel

To create that type of channels the implementation is exactly the same than in the [previous section](#). So in this section we are going to show only the code differences.

The first app will use the method `createAsyncRequestReplyChannel` to create the channel, the rest is the same

```
IntegrationChannel demoAsyncChannel =
    this.integration.createAsyncRequestReplyChannel("my-async-channel", "my-async-queue", this.myIntegrationHandler);
```

You can define your own values for the *ThreadPoolExecutor's core pool size* and *response timeout* adding the properties `devonfw.integration.default.poolsize` and `devonfw.integration.default.receiveTimeout` to your `application.properties` file and providing a value.

However, you can also define those values when creating the channel

```
IntegrationChannel demoAsyncChannel =
    this.integration.createAsyncRequestReplyChannel("my-async-channel", "my-async-queue", this.myMessageHandler, 15, 10000);
```

In the second app you can subscribe to the channel with the method `subscribeAndReplyAsyncTo` and providing the names for the channel and queue (that **must match** with the names provided in the first application), and an *IntegrationHandler* to manage the messages and provide a reply.

```
this.integration.subscribeAndReplyAsyncTo("my-async-channel", "my-async-queue", this.myIntegrationHandler);
```

The implementation for the *Integration Handler* is explained [here](#).

[[cookbook-integration-module\_sending-headers with the message]] === Sending headers with the message

The *Integration module* also allows to send headers alongside the message content. To do so you can use the methods provided by the module that accept a *Map* as parameter for headers.

[[cookbook-integration-module\_creating-the headers]] ===== Creating the headers

You can create the message headers using a Java *Map* object

```
Map headers = new HashMap();
headers.put("header1", "value1");
headers.put("header2", "value2");
```

[[cookbook-integration-module\_sending-the headers]] ===== Sending the headers

Each *send* method provided with the module accepts a *Map* object as parameter for the headers, so you can send it alongside the message content

- **default simple channel:** `integration.send("Hello", headers)`
- **default request-reply channel:** `integration.sendAndReceive("Hello", headers)`
- **default asynchronous request-reply channel:**  
`integration.sendAndReceiveAsync("Hello", headers)`
- **new created channels:** `new_channel.send("Hello", headers)`

## 3.8 Compose for Redis module

### 3.8.1 Introduction

[Redis](#) is an open-source, blazingly fast, key/value low maintenance store. [Compose](#)'s platform gives you a configuration pre-tuned for high availability and locked down with additional security features. The component will manage the service connection and the main methods to manage the key/values on the storage.

[[cookbook-compose-for-redis-module\_include-compose for redis in a devon project]] == Include Compose for Redis in a Devon project

The module provides you a connection to a compose for redis service, and a bunch of predefined operations for managing the objects inside the service, for your Devon applications. To implement the module in a Devon project, you must follow these steps:

[[cookbook-compose-for-redis-module\_step-1: adding the dependency in your project]] === Step 1: Adding the dependency in your project

Include the module dependency in your pom.xml, verifying that the *version* matches the last available version of the module

```
<dependency>
  <groupId>com.capgemini.devonfw.modules</groupId>
  <artifactId>devonfw-compose-redis</artifactId>
  <version>2.2.0</version>
</dependency>
```

#### Warning

The IP modules of Devonfw are stored in [Artifactory](#). In case, you do not have access to that repository, as the modules are included in the Devonfw distribution, you can install them manually. To do so, open a Devonfw command line (*console.bat*), go to *Devon-dist\workspaces\examples\devon\modules* and execute the command *mvn install*. If the project is already imported in Eclipse then update project: Right click on *project > Maven > Update Project* > check the *Force update of Snapshot/Releases* checkbox > *Ok*

[[cookbook-compose-for-redis-module\_step-2: properties configuration]] === Step 2: Properties configuration

In order to use the module for managing the cache service, it is necessary to define some connection parameters in the application.properties file in the project.

```
# Compose for Redis module params
devon.redis.service.name = compose-for-redis
devon.redis.uri = redis://admin:RFAAVWWZSXXXXX@sl-eu-lon-2-portal.1.dblayer.com:16552
devon.redis.runTests.enable = false
```

- The parameter `devon.redis.service.name` is used to lookup the service in a cloud environment from the environment variable information `VCAP_SERVICES`. Default value `compose-for-redis`.
- The parameter `devon.redis.uri` is used to indicate the service's uri in case we need to connect to the service outside of a cloud environment. There is no default value.
- The parameter `devon.redis.runTests.enable` is used for enabling the test execution. By default this execution is disabled.

**Note**

- When the parameter `devon.redis.service.name` is indicated, and the application is running on a cloud environment, the parameter `devon.redis.uri` will be never used.
- The parameter `devon.redis.uri` is used when there is no value for the parameter `devon.redis.service.name` or when the application is not running on a cloud environment.
- In case the test execution is enabled (true) the service's uri (`devon.redis.uri`) must be indicated because the test execution context will be outside a cloud environment.

### 3.8.2 Basic implementation

[[cookbook-compose-for-redis-module\_the-injection of the module]] === The injection of the module

To access the module functionalities, you need to inject the module in a private property, it can be done using the `@Inject` annotation

```
public class MyClass {  
  
    @Inject  
    LettuceManagement lettuceManagement;  
  
    [...]  
}
```

The `lettuceManagement` object will give us access to all the module functionalities described in the following section.

### 3.8.3 Available operations

```
/**  
 * Set a hash entry  
 * @param hashName hash name  
 * @param key key  
 * @param value value  
 * @return true if the entry has been set, false otherwise  
 */  
boolean setHashEntry(String hashName, String key, String value);  
  
/**  
 * Get a hash entry  
 * @param hashName hash name  
 * @param key key  
 * @return value  
 */  
String getHashEntry(String hashName, String key);  
  
/**  
 * Get a hash map  
 * @param hashName hash name  
 * @return map  
 */  
Map<String, String> getHash(String hashName);  
  
/**  
 * Check if a Hash Entry exists  
 * @param hashName hash name  
 * @param key key  
 * @return true if it exists, false otherwise  
 */  
Boolean hashEntryExists(String hashName, String key);
```

```
/**  
 * Set a hash map  
 * @param hashName The name for the map  
 * @param map The map object to be persisted on Redis  
 * @return simple-string-reply  
 */  
String setHash(String hashName, Map<String, String> map);  
  
/**  
 * Delete hash map entries  
 * @param hashName The name for the map  
 * @param fields Field names to be deleted  
 * @return True if all the given fields has been deleted, false otherwise  
 */  
Boolean deleteHashEntries(String hashName, String... fields);
```

## 3.9 Security

Security is todays most important cross-cutting concern of an application and an enterprise IT-landscape. We seriously care about security and give you detailed guides to prevent pitfalls, vulnerabilities, and other disasters. While many mistakes can be avoided by following our guidelines you still have to consider security and think about it in your design and implementation. The security guides provided by this document will not automatically prevent you from any harm, but they may give you hints and best practices already used in different software products.

### 3.9.1 Authentication

Definition:

Authentication is the verification that somebody interacting with the system is the actual subject for whom he claims to be.

The one authenticated is properly called *subject* or *principal*. However, for simplicity we use the common term *user* even though it may not be a human (e.g. in case of a service call from an external system).

To prove his authenticity the user provides some secret called *credentials*. The most simple form of credentials is a password.

#### Note

Please never implement your own authentication mechanism or credential store. You have to be aware of implicit demands such as salting and hashing credentials, password life-cycle with recovery, expiry, and renewal including email notification confirmation tokens, central password policies, etc. This is the domain of access managers and identity management systems. In a business context you will typically already find a system for this purpose that you have to integrate (e.g. via LDAP).

oasp4j uses Spring Security as a framework for authentication purposes. In-memory authentication for an user is configured using Spring security. Refer the overriden *configureGlobal* method of *AbstractWebSecurityConfig.java* of OASP4J for an implementation of In-memory authentication.

```
@SuppressWarnings("javadoc")
@.Inject
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");
}
```

#### Mechanisms

##### Basic

Http-Basic authentication can be easily implemented using *BasicAuthenticationFilter*:

##### Form Login

For a form login the spring security implementation might look like this:

```
.formLogin().successHandler(new SimpleUrlAuthenticationSuccessHandler()).defaultSuccessUrl("/")
    .failureUrl("/login.html?error").loginProcessingUrl("/")
    .springSecurityLogin().usernameParameter("username")
    .passwordParameter("password").and()
    .logout().logoutSuccessUrl("/login.html")
```

The interesting part is, that there is a login-processing-url, which should be addressed to handle the internal spring security authentication and similarly there is a logout-url, which has to be called to logout a user.

[[cookbook-security-layer\_preserve-original request anchors after form login redirect]] === Preserve original request anchors after form login redirect

Spring Security will automatically redirect any unauthorized access to the defined login-page. After successful login, the user will be redirected to the original requested URL. The only pitfall is, that anchors in the request URL will not be transmitted to server and thus cannot be restored after successful login. Therefore the oasp4j-security module provides the RetainAnchorFilter, which is able to inject javascript code to the source page and to the target page of any redirection. Using javascript this filter is able to retrieve the requested anchors and store them into a cookie. Heading the target URL this cookie will be used to restore the original anchors again.

To enable this mechanism you have to integrate the RetainAnchorFilter as follows:

First, declare the filter with:

- `storeUrlPattern` a regular expression matching the URL, where anchors should be stored
- `restoreUrlPattern` a regular expression matching the URL, where anchors should be restored
- `cookieName` the name of the cookie to save the anchors in the intermediate time

```
@Bean
public RetainAnchorFilter retainAnchorFilter() {

    RetainAnchorFilter retainAnchorFilter = new RetainAnchorFilter();

    // first [^/]+ part describes host name and possibly port, second [^/]+ is the application name
    retainAnchorFilter.setStoreUrlPattern("http://[^/]+/[^\+]+/login.*");
    retainAnchorFilter.setRestoreUrlPattern("http://[^/]+/[^\+]+/.*");
    retainAnchorFilter.setCookieName("TARGETANCHOR");
    return retainAnchorFilter;
}
```

Second, register the filter as first filter in the request filter chain. You might want to use the `before="FIRST"` or `after="FIRST"` attribute if you have multiple request filters, which should be run before the default filters.

### simple Spring Security filter insertion.

```
.addFilterAfter(FIRST, RetainAnchorFilter.class)
```

Nevertheless, the oasp4j follows a different approach. The simple interface of Spring Security for inserting custom filters as stated above is driven by a relative alignment of the different filters been executed. You relatively can insert custom filters before or after existing ones and also at the beginning or at the end. You might easily see, that the real filter chain will get more and more invisible. Thus the oasp4j follows the default ordering of the Spring Security filter chain, such that it gets more transparent for any developer, which filters will be executed in which order and at which position a new custom filter may be inserted.

This documentation depends on Spring Security v4.2.3.RELEASE:

- <http://docs.spring.io/spring-security/site/docs/4.2.3.BUILD-SNAPSHOT/reference/htmlsingle/#ns-custom-filters>

These lists will be maintained each release, which will include a Spring Security upgrade. Thus first, we will not lose any changes from the possibly updated default filter chain of Spring Security. Second, due to the absolute declaration of the filter order, you might not get any strange behavior in your system after upgrading to a new version of Spring Security.

[[cookbook-security-layer\_users-vs. systems]] === Users vs. Systems If we are talking about authentication we have to distinguish two forms of principals:

- human users
- autonomous systems

While e.g. a Kerberos/SPNEGO Single-Sign-On makes sense for human users it is pointless for authenticating autonomous systems. So always keep this in mind when you design your authentication mechanisms and separate access for human users from access for systems.

### 3.9.2 Authorization

#### Definition:

Authorization is the verification that an authenticated user is allowed to perform the operation he intends to invoke.

[[cookbook-security-layer\_clarification-of terms]] === Clarification of terms

For clarification we also want to give a common understanding of related terms that have no unique definition and consistent usage in the wild.

Table 3.2. Security terms related to authorization

Term	Meaning and comment
Permission	A permission is an object that allows a principal to perform an operation in the system. This permission can be <i>granted</i> (give) or <i>revoked</i> (taken away). Sometimes people also use the term <i>right</i> what is actually wrong as a right (such as the right to be free) can not be revoked.
Group	We use the term group in this context for an object that contains permissions. A group may also contain other groups. Then the group represents the set of all recursively contained permissions.
Role	We consider a role as a specific form of group that also contains permissions. A role identifies a specific function of a principal. A user can act in a role.  For simple scenarios a principal has a single role associated. In more complex situations a principal can have multiple roles but has only one active role at a time that he can choose out of his assigned roles. For KISS it is sometimes sufficient to avoid this by creating multiple accounts for the few users with multiple roles. Otherwise at least avoid switching roles at runtime in clients as this may cause problems with related states. Simply restart the client with the new role as parameter in case the user wants to switch his role.
Access Control	Any permission, group, role, etc., which declares a control for access management.

[[cookbook-security-layer\_suggestions-on the access model]] === Suggestions on the access model  
 The access model provided by oasp4j-security follows this suggestions:

- Each Access Control (permission, group, role, ...) is uniquely identified by a human readable string.
- We create a unique permission for each use-case.
- We define groups that combine permissions to typical and useful sets for the users.
- We define roles as specific groups as required by our business demands.
- We allow to associate users with a list of Access Controls.
- For authorization of an implemented use case we determine the required permission. Furthermore, we determine the current user and verify that the required permission is contained in the tree spanned by all his associated Access Controls. If the user does not have the permission we throw a security exception and thus abort the operation and transaction.
- We try to avoid negative permissions, that is a user has no permission by default but only those granted to him additively permit him for executing use cases.
- Technically we consider permissions as a secret of the application. Administrators shall not fiddle with individual permissions but grant them via groups. So the access management provides a list of strings identifying the Access Controls of a user. The individual application itself contains these Access Controls in a structured way, whereas each group forms a permission tree.

### **oasp4j-security**

The OASP4J applications provide a ready to use module [oasp4j-security](#) that is based on [spring-security](#) and makes your life a lot easier.

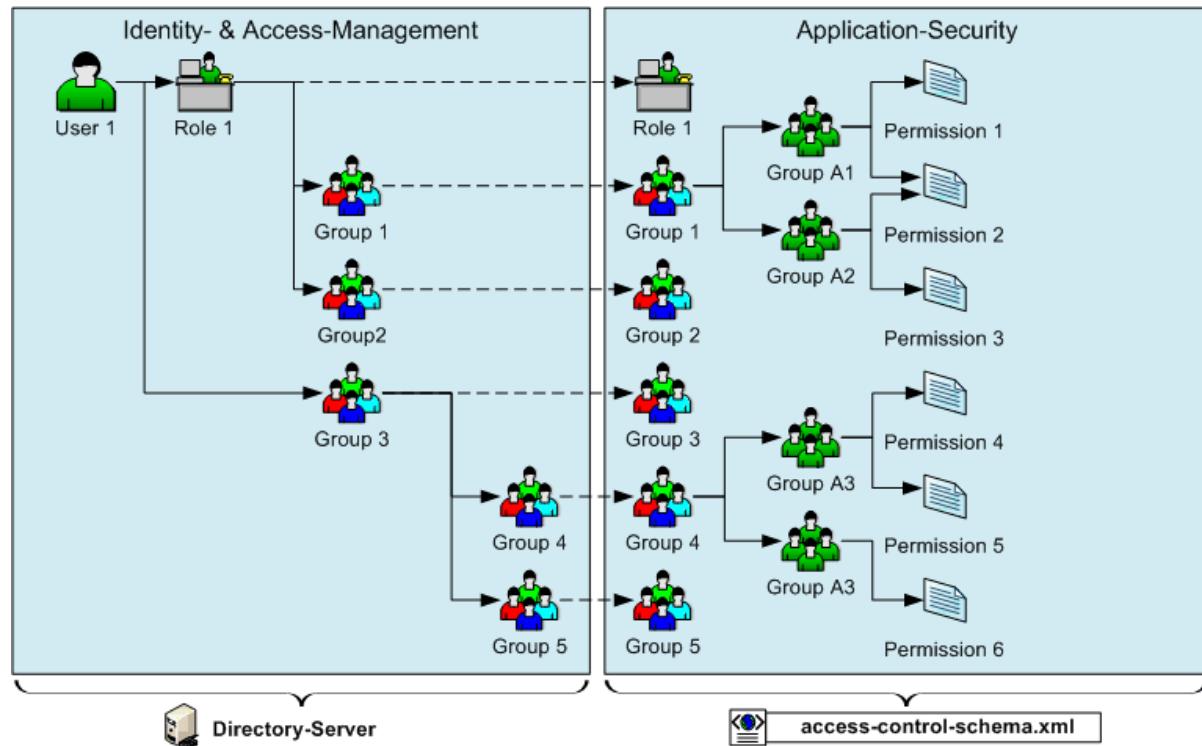


Figure 3.1. OASP4J Security Model

The diagram shows the model of *oasp4j-security* that separates two different aspects:

- The *Identity- and Access-Management* is provided by according products and typically already available in the enterprise landscape (e.g. an active directory). It provides a hierarchy of *primary access control objects* (roles and groups) of a user. An administrator can grant and revoke permissions (indirectly) via this way.
- The application security is using *oasp4j-security* and defines a hierarchy of *secondary access control objects* (groups and permissions) in the file `access-control-schema.xml` (see [example from sample app](#)). This hierarchy defines the application internal access control schema that should be an implementation secret of the application. Only the top-level access control objects are public and define the interface to map from the primary to secondary access control objects. This mapping is simply done by using the same names for access control objects to match.

[[cookbook-security-layer\_access-control schema]] ===== Access Control Schema

The `oasp4j-security` module provides a simple and efficient way to define permissions and roles. The file `access-control-schema.xml` is used to define the mapping from groups to permissions. The general terms discussed above can be mapped to the implementation as follows:

Table 3.3. General security terms related to *oasp4j* access control schema

Term	<b><i>oasp4j-security implementation</i></b>	Comment
Permission	<code>AccessControlPermission</code>	
Group	<code>AccessControlGroup</code>	When considering different levels of groups of different meanings, declare <code>type</code> attribute, e.g. as "group".
Role	<code>AccessControlGroup</code>	With <code>type="role"</code> .
Access Control	<code>AccessControl</code>	Super type that represents a tree of <code>AccessControlGroups</code> and <code>AccessControlPermissions</code> . If a principal "has" a <code>AccessControl</code> he also "has" all <code>AccessControls</code> with according permissions in the spanned sub-tree.

### Example `access-control-schema.xml`.

```

<?xml version="1.0" encoding="UTF-8"?>
<access-control-schema>
  <group id="ReadMasterData" type="group">
    <permissions>
      <permission id="OfferManagement_GetOffer"/>
      <permission id="OfferManagement_GetProduct"/>
      <permission id="TableManagement_GetTable"/>
      <permission id="StaffManagement_GetStaffMember"/>
    </permissions>
  </group>

  <group id="Waiter" type="role">
    <inherits>
      <group-ref>Barkeeper</group-ref>
    </inherits>
    <permissions>
      <permission id="TableManagement_ChangeTable"/>
    </permissions>
  </group>
</access-control-schema>

```

```
</group>
...
</access-control-schema>
```

This example `access-control-schema.xml` declares

- a group named `ReadMasterData`, which grants four different permissions, e.g., `OfferManagement_GetOffer`
- a group named `Waiter`, which
  - also grants all permissions from the group `Barkeeper`
  - in addition grants the permission `TableManagement_ChangeTable`
  - is marked to be a `role` for further application needs.

The `oasp4j-security` module automatically validates the schema configuration and will throw an exception if invalid.

The permissions and roles defined in `access-control-schema.xml` are loaded using implementation of `AccessControlSchemaProvider` interface as show below.

```
InputStream inputStream = this.accessControlSchema.getInputStream();
AccessControlSchema schema = this.accessControlSchemaMapper.read(inputStream);
```

Group permissions are collected that principal is part of and is used to decide access to application resources as shown below

```
Set<GrantedAuthority> authorities = new HashSet<>();
Collection<String> accessControlIds
= this.principalAccessControlProvider.getAccessControlIds(principal);
Set<AccessControl> accessControlSet = new HashSet<>();
for (String id : accessControlIds) {
    boolean success = this.accessControlProvider.collectAccessControls(id, accessControlSet);
    if (!success) {
        LOG.warn("Undefined access control {}.", id);
    }
}
for (AccessControl accessControl : accessControlSet) {
    authorities.add(new AccessControlGrantedAuthority(accessControl));
}
```

[[cookbook-security-layer\_configuration-on url level]] ===== Configuration on URL level

The authorization (in terms of Spring security "access management") can be enabled separately for different url patterns, the request will be matched against. The order of these url patterns is essential as the first matching pattern will declare the access restriction for the incoming request (see `access` attribute). Here an example:

**Extensive example of authorization on URL level.**

```
.authorizeRequests()
    .antMatchers("/login", "/home").permitAll()
    .antMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()
```

[[cookbook-security-layer\_configuration-on java method level]] ===== Configuration on Java Method level As state of the art oasp4j will focus on role-based authorization to cope with authorization for executing use case of an application. We will use the `JSR250` annotations, mainly `@RolesAllowed`,

for authorizing method calls against the permissions defined in the annotation body. This has to be done for each use-case method in logic layer. Here is an example:

```
@RolesAllowed(PermissionConstants.FIND_TABLE)
public TableEto findTable(Long id) {
    return getBeanMapper().map(getTableDao().findOne(id), TableEto.class);
}
```

Now this method *findTable* can only be called if the user that is logged-in has the permission `FIND_TABLE`.

More in detail, imagine that you have two types of users in your app: *customers* and *admins*. So you want to allow both of them to see your products but only *admins* can create new ones. In the Devonfw based apps ( *JSR250* annotations) the way you should proceed to achieve that would be

1 - Define the roles in the `access-control-schema.xml` file (usually located on `/src/main/resources/config/app/security`)

```
<?xml version="1.0" encoding="UTF-8"?>
<access-control-schema>
    ...
    <group id="Customer" type="role">
        <permissions>
            <permission id="SeeProducts"/>
        </permissions>
    </group>

    <group id="Admin" type="role">
        <inherits>
            <group-ref>Customer</group-ref>
        </inherits>
        <permissions>
            <permission id="CreateProduct"/>
        </permissions>
    </group>
    ...
</access-control-schema>
```

As you can see we have created two roles *Customer* and *Admin*. The *Customer* can *SeeProducts* and the *Admin* inherits permissions from *Customer*, so he can also *SeeProducts*, and in addition to that we have defined an new permission *CreateProduct* to allow only him to create new products.

2 - Is recommended, although optional, to use an intermediate class to manage the permission terminology to avoid errors. So we could create a class *PermissionConstants* and store there the permission names. You can use [Cobigen](#) to easily generate this class.

```
public abstract class PermissionConstants {

    public static final String SEE_PRODUCTS = "SeeProducts";

    public static final String CREATE_PRODUCT = "CreateProduct";

    ...
}
```

3 - Finally, in our theoretical *Productmanagement* use case, at implementation level (`src/main/java/my/devonfw/app/productmanagement/logic/impl/ProductmanagementImpl.java`), we can define the permissions for each method using the `@RolesAllowed` annotation

```
public class ProductmanagementImpl extends AbstractComponentFacade implements Productmanagement {
```

```
@Override
@RolesAllowed(PermissionConstants.SEE_PRODUCTS)
public ProductEto findProduct(Long id) {

    ProductEntity product = getProductDao().load(id);
    return getBeanMapper().map(product, ProductEto.class);
}

@Override
@RolesAllowed(PermissionConstants.CREATE_PRODUCT)
public ProductEto saveProduct(ProductEto product) {

    Objects.requireNonNull(product, "product");

    ProductEntity persistedProduct = getProductDao().save(getBeanMapper().map(product,
ProductEntity.class));
    return getBeanMapper().map(persistedProduct, ProductEto.class);
}

}
```

At this point, with these three simple steps, the regular customers can see the products but not create new ones, while *admins* can do both operations.

[[cookbook-security-layer\_check-data-based permissions]] ===== Check Data-based Permissions  
Currently, we have no best practices and reference implementations to apply permission based access on an application's data. Nevertheless, this is a very important topic due to the high standards of data privacy & protection especially in germany. We will further investigate this topic and we will address it in one of the next releases. For further tracking have a look at [issue #125](#).

[[cookbook-security-layer\_spring-security apis and their usage in oasp4j]] === Spring Security APIs and their usage in OASP4j

### **UsersDetailsService**

*UserDetailsService* is a core interface used to load user-specific data. It has `loadUserByUsername()` method to find a user entity and can be overridden to provide custom implementation. For further reading follow the Spring Security documentation [here](#)

### **BaseUserDetailsService**

This is a custom implementation of Spring's *UserDetailsService*. It overrides `loadUserByUsername()` method and returns *UserDetails* with user data and authorities. This implementation of *UserDetailsService* is further used in *AbstractWebSecurityConfig*.

### **AbstractWebSecurityConfig**

This class extends Spring Security's class *WebSecurityConfigurerAdapter* and overrides `configure()` method. Here *BaseUserDetailsService* is set in Spring *HttpSecurity* that configures web based security for http requests.

[[cookbook-security-layer\_vulnerabilities-and protection]] == Vulnerabilities and Protection Independent from classical authentication and authorization mechanisms there are many common pitfalls that can lead to vulnerabilities and security issues in your application such as XSS, CSRF, SQL-injection, log-forging, etc. A good source of information about this is the [OWASP](#). We address these common threats individually in *security* sections of our technological guides as a concrete solution to prevent an attack typically depends on the according technology. The following table illustrates common threats and contains links to the solutions and protection-mechanisms provided by the OASP:

Table 3.4. Security threats and protection-mechanisms

Thread	Protection	Link to details
<a href="#">A1 Injection</a>	validate input, escape output, use proper frameworks	<a href="#">data-access-layer guide</a>
<a href="#">A2 Broken Authentication and Session Management</a>	encrypt all channels, use a central identity management with strong password-policy	<a href="#">Authentication</a>
<a href="#">A3 XSS</a>	prevent injection (see A1) for HTML, JavaScript and CSS and understand same-origin-policy	<a href="#">client-layer</a>
<a href="#">A4 Insecure Direct Object References</a>	Using direct object references (IDs) only with appropriate authorization	<a href="#">logic-layer</a>
<a href="#">A5 Security Misconfiguration</a>	Use OASP application template and guides to avoid	<a href="#">application template</a>
<a href="#">A6 Sensitive Data Exposure</a>	Use secured exception facade, design your data model accordingly	<a href="#">REST exception handling</a>
<a href="#">A7 Missing Function Level Access Control</a>	Ensure proper authorization for all use-cases, use <code>@DenyAll</code> as default to enforce	<a href="#">Method authorization</a>
<a href="#">A8 CSRF</a>	secure mutable service operations with an explicit CSRF security token sent in HTTP header and verified on the server	<a href="#">service-layer security</a>
<a href="#">A9 Using Components with Known Vulnerabilities</a>	subscribe to security newsletters, recheck products and their versions continuously, use OASP dependency management	<a href="#">CVE newsletter</a>
<a href="#">A10 Unvalidated Redirects and Forwards</a>	Avoid using redirects and forwards, in case you need them do a security audit on the solution.	OASP proposes to use rich-clients (SPA/RIA). We only use redirects for login in a safe way.
<a href="#">Log-Forging</a>	Escape newlines in log messages	<a href="#">logging security</a>

Tool for testing your web application against vulnerabilities: [OWASP Zed Attack Proxy Project](#)

1. Easy to Install
2. Supports Different types of Fuzzer Based Tests

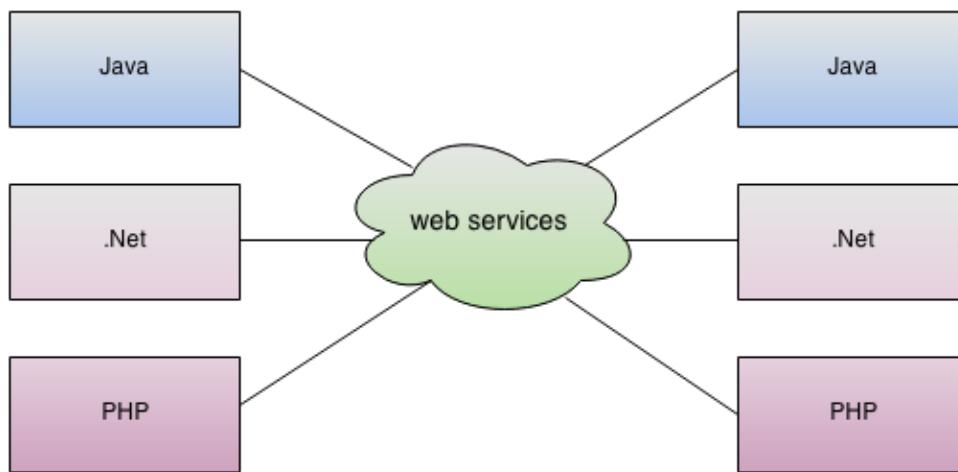
- 
- 3. Details Results Reports
  - 4. Convenient to carry out Test on Staging environment

## 3.10 Web Services (JAX WS)

[[cookbook-jax-ws-webservices\_what-are web services]] == What are Web Services

A Web service can be defined by following ways:

- a client-server application or an application component for the communication.
- a method of communication between the two devices over network.
- a software system for interoperable machine to machine communication.
- a collection of standards or protocols for exchanging information between the two devices or an application.



In the above figure, java, .net or PHP applications can communicate with other applications through the web service over the network. For example, a java application can interact with Java, .Net and PHP applications. Hence, the web service is a language independent way of communication.

There are mainly two types of web services:

1. SOAP web services.
2. RESTful web services.

Next section describes more about JAX-WS (SOAP based) web services.

[[cookbook-jax-ws-webservices\_why-use web services]] == Why use Web Services

### Exposing the Existing Function on the network

A web service is a unit of managed code that can be remotely invoked using HTTP, that is, it can be activated using HTTP requests. Web services allow you to expose the functionality of your existing code over the network. Once it is exposed on the network, other applications can use the functionality of your program.

### Interoperability

Web services allow various applications to talk to each other and share data and services among themselves. Other applications can also use the web services. For example, a VB or .NET application

---

can talk to Java web services and vice versa. Web services are used to make the application, platform and technology independent.

### **Standardized Protocol**

Web services use standardized industry standard protocol for the communication. All the four layers (Service Transport, XML Messaging, Service Description, and Service Discovery layers) use well-defined protocols in the web services protocol stack. This standardization of protocol stack gives the business many advantages, such as a wide range of choices, reduction in the cost due to competition, and increase in the quality.

### **Low Cost of Communication**

Web services use SOAP over HTTP protocol, so you can use your existing low-cost internet for implementing web services. This solution is much less costly compared to the proprietary solutions like EDI/B2B. Besides SOAP over HTTP, web services can also be implemented on the other reliable transport mechanisms like FTP.

[[cookbook-jax-ws-webservices\_characteristics-of web services]] == Characteristics of Web Services

### **XML-Based**

Web Services uses XML at data representation and data transportation layers. Using XML, eliminates any networking, operating system, or platform binding. Web Services based applications are highly interoperable application at their core level.

### **Loosely Coupled**

A consumer of a web service is not tied to that web service directly. The web service interface can change over time without compromising the client's ability to interact with the service. A tightly coupled system implies that the client and server logic are closely tied to one another, implying that if one interface changes, the other must be updated. Adopting a loosely coupled architecture tends to make software systems more manageable and allows simpler integration between different systems.

### **Coarse-Grained**

Object-oriented technologies such as Java expose their services through individual methods. An individual method is a too fine operation to provide any useful capability at a corporate level. Building a Java program from scratch requires the creation of several fine-grained methods that are then composed into a coarse-grained service that is consumed by either a client or the other service.

Businesses and the interfaces that they expose should be coarse-grained. Web services technology provides a natural way of defining coarse-grained services that access the right amount of business logic.

### **Ability to be Synchronous or Asynchronous**

Synchronicity refers to the binding of the client to the execution of the service. In synchronous invocations, the client blocks and waits for the service to complete its operation before continuing. Asynchronous operations allow a client to invoke a service and then execute other functions.

Asynchronous clients retrieve their results at a later point in time, while synchronous clients receive their results, when the service has completed. Asynchronous capability is a key factor in enabling loosely coupled systems.

---

[[cookbook-jax-ws-webservices\_components-of soap based web service]] == Components of SOAP based Web Service

There are three major web service components:

1. SOAP
2. WSDL
3. UDDI

### **SOAP**

SOAP is an acronym for Simple Object Access Protocol.

SOAP is an XML-based protocol for accessing web services.

SOAP is a W3C recommendation for communication between applications.

SOAP is XML based, so it is platform independent and language independent. In other words, it can be used with Java, .Net or PHP language on any platform.

### **WSDL**

WSDL is an acronym for Web Services Description Language.

WSDL is an xml document containing information about web services such as method name, method parameter and how to access it.

WSDL is a part of UDDI. It acts as an interface between web service applications.

WSDL is pronounced as "wiz-dull".

### **UDDI**

UDDI is an acronym for Universal Description, Discovery and Integration.

UDDI is an XML based framework for describing, discovering and integrating web services.

UDDI is a directory of web service interfaces described by WSDL, containing information about web services.

[[cookbook-jax-ws-webservices\_apache-cxf and jax ws ]] == Apache CXF and JAX WS

CXF implements the JAX-WS APIs which makes building web services easier. JAX-WS encompasses many different areas:

- Generating WSDL from Java classes and generating Java classes from WSDL
- Provider API which allows you to create simple messaging receiving server endpoints
- Dispatch API which allows you to send raw XML messages to server endpoints
- Spring integration
- It supports Restful services too

In devonfw, Apache CXF implementation of JAX WS is used.

[[cookbook-jax-ws-webservices\_creation-of web service using apache cxf]] == Creation of Web Service using Apache CXF

## Developing the service

This can be done in two ways: *code-first* and *contract-first*. The *code-first* approach is used below:

Here is an example in case you define a *code-first* service. Create a regular interface to define the API of the service and annotate it with JAX-WS annotations:

```
@WebService
public interface TablemanagementWebService {

    @WebMethod
    @WebResult(name = "message")
    TableEto getTable(@WebParam(name = "id") String id);

}
```

And here is a simple implementation of the service:

```
@Named( "TablemanagementWebService" )
@WebService(endpointInterface =
"io.oasp.gastronomy.restaurant.tablemanagement.service.api.ws.TablemanagementWebService")
public class TablemanagementWebServiceImpl implements TablemanagementWebService {

    private Tablemanagement tableManagement;

    @Override
    public TableEto getTable(String id) {

        return this.tableManagement.findTable(id);
    }
}
```

If you look at the above interface, you can tell that it is a normal Java interface with the exception of three annotations:

- **@WebService** – Specifies that the JWS file implements a web service turning a normal [POJO](#) into a web service. In the above case, the annotation is placed right above the interface definition and it notifies that TablemanagementWebService is not a normal interface rather an web service interface or SEI.
- **@WebMethod** – This annotation is optional and is mainly used to provide a name attribute to the public method in WSDL.
- **@WebResult** - The **@WebResult** annotation allows you to specify the properties of the generated wsdl:part that is generated for the method's return value.
- **@WebParam** - The **@WebParam** annotation is defined by the javax.jws.WebParam interface. It is placed on the parameters on the methods defined in the SEI. The **@WebParam** annotation allows you to specify the direction of the parameter, if the parameter will be placed in the SOAP header, and other properties of the generated wsdl:part.

The **@WebService** annotation on the implementation class lets CXF know which interface to use when creating WSDL. In this case, it's simply our *TablemanagementWebService* interface.

Finally, you have to register the service implementation in the spring in this **@Configuration**-annotated Class. Here, the CXF and end point is initialized. So, the **@Configuration**-annotated

Class that is *ServiceConfiguration.java* can be found within the sample app in `src/main/java/io.oasp.gastronomy.restaurant/general/configuration` of `xxx-core` project.

```
@Configuration
@EnableWs
@ImportResource({ "classpath:META-INF/cxf/cxf.xml" /* , "classpath:META-INF/cxf/cxf-servlet.xml" */ })
public class ServiceConfiguration extends WsConfigurerAdapter {

    @Bean(name = "cxf")
    public SpringBus springBus() {

        return new SpringBus();
    }

    @Bean
    public ServletRegistrationBean servletRegistrationBean() {

        CXFServlet cxfServlet = new CXFServlet();
        ServletRegistrationBean servletRegistration = new ServletRegistrationBean(cxfServlet,
        URL_PATH_SERVICES + "/");
        return servletRegistration;
    }

    // BEGIN ARCHETYPE SKIP
    @Bean
    public Endpoint tableManagement() {

        EndpointImpl endpoint = new EndpointImpl(springBus(), new TablemanagementWebServiceImpl());
        endpoint.publish("/TablemanagementWebService");
        return endpoint;
    }
    // END ARCHETYPE SKIP
}
```

You can see the beans *SpringBus* and *ServletRegistrationBean* inside the `@Configuration`-Class. You need to configure it to return an instance of `org.apache.cxf.jaxws.EndpointImpl`, which later will be forwarded to the *SpringBus* and the implementor via constructor-arg:

Furthermore, you have to use the `publish` method of the `org.apache.cxf.jaxws.EndpointImpl` to define the last part of the WebService-URI.

Now, if you are fire up the sample application with [SpringBoot](#), opening a browser and hit below URL where the web service is hosted:

```
http://localhost:8081/oasp4j-sample-server/services/
```

You should see our *TablemanagementService* beneath "Available SOAP services" including all available web service methods.

[[cookbook-jax-ws-webservices\_soap-custom mapping]] == Soap Custom Mapping

In order to map custom [datatypes](#) or other types that do not follow the Java bean conventions, you need to write the adapters for JAXB [XML](#).

### 3.10.1 SOAP Testing

For testing SOAP services manually, it is strongly recommended to use [SoapUI](#).

## 3.11 Batch Layer

We understand batch processing as bulk-oriented, non-interactive, typically long running execution of tasks. For simplicity we use the term batch or batch job for such tasks in the following documentation.

OASP4J uses [Spring Batch](#) as batch framework.

This guide explains how Spring Batch is used in OASP4J applications. Please note that it is not yet, fully consistent concerning batches with the sample application. You should adhere to this guide for now.

### 3.11.1 Batch architecture

In this chapter we will describe the overall architecture (especially concerning layering) and how to administer batches.

#### Layering

Batches are implemented in the batch layer. The batch layer is responsible for batch processes, whereas the business logic is implemented in the logic layer. Compared to the [service layer](#) you may understand the batch layer just as a different way of accessing the business logic. From a component point of view each batch is implemented as a subcomponent in the corresponding business component. The business component is defined by the [business architecture](#).

Let's make an example for that. The sample application implements a batch for exporting bills. This bill-export-batch belongs to the sales management business component. So the bill-export-batch is implemented in the following package:

```
<basepackage>.salesmanagement.batch.impl.billexport.*
```

Batches should invoke use cases in the logic layer for doing their work. Only "batch specific" technical aspects should be implemented in the batch layer.

Example: For a batch, which imports product data from a CSV file this means that all code for actually reading and parsing the CSV input file is implemented in the batch layer. The batch calls the use case "create product" in the logic layer for actually creating the products for each line read from the CSV input file.

#### Accessing data access layer

In practice it is not always appropriate to create use cases for every bit of work a batch should do. Instead, the data access layer can be used directly. An example for that is a typical batch for data retention which deletes out-of-time data. Often deleting out-dated data is done by invoking a single SQL statement. It is appropriate to implement that SQL in a [DAO](#) method and call this method directly from the batch. But be careful that this pattern is a simplification which could lead to business logic cluttered in different layers which reduces maintainability of your application. It is a typical design decision you have to take when designing your specific batches.

#### Batch administration and execution

##### Starting and Stopping Batches

Spring Batch provides a simple command line API for execution and parameterization of batches, the `CommandLineJobRunner`. However, it is not yet fully compatible with Spring Boot. For those using Spring Boot OASP4J provides the `SpringBootBatchCommandLine` with similar functionalities.

Both execute batches as a "simple" standalone process (instantiating a new JVM and creating a new *ApplicationContext*).

### Starting a Batch Job

For starting a batch job, the following parameters are required:

#### jobPath

The location of the JavaConfig classes (usually annotated with `@Configuration` or `@SpringBootApplication`) and/or XML files that will be used to create an *ApplicationContext*.

The `CommandLineJobRunner` only accepts one class/file, which must contain everything needed to run a job (potentially by referencing other classes/files), the `SpringBootBatchCommandLine`, however, expects that there are two paths given: one for the general batch setup and one for the XML file containing the batch job to be executed.

There is an example of a general batch setup for Spring Boot in the samples/core (`oasp4j-sample-core`) project, class `SpringBootBatchApp`, which also imports the general configuration class introduced in the chapter on the [general configuration](#). Note that `SpringBootBatchApp` deactivates the evaluation of annotations used for authorization, especially the `@RolesAllowed` annotation. You should of course make sure that only authorized users can start batches, but once the batch is started there is usually no need to check any authorization.

#### jobName

The name of the job to be run.

All arguments after the job name are considered to be job parameters and must be in the format of 'name=value':

Example for the `CommandLineJobRunner`:

```
java org.springframework.batch.core.launch.support.CommandLineJobRunner classpath:config/app/batch/
beans-billexport.xml billExportJob -outputFile=file:out.csv date(date)=2015/12/20
```

Example for the `SpringBootBatchCommandLine`:

```
java io.oasp.module.batch.common.base.SpringBootBatchCommandLine
io.oasp.gastronomy.restaurant.SpringBootBatchApp classpath:config/app/batch/beans-billexport.xml
billExportJob -outputFile=file:out.csv date(date)=2015/12/20
```

The date parameter will be explained in the section on [parameters](#).

Note that when a batch is started with the same parameters as a previous execution of the same batch job, the new execution is considered a restart, see [restarts](#) for further details. Parameters starting with a "-" are ignored when deciding whether an execution is a restart or not (so called non identifying parameters).

When trying to restart a batch that was already complete, there will either be an exception (message: "A job instance already exists and is complete for parameters={...}. If you want to run this job again, change the parameters.") or the batch will simply do nothing (might happen when no or only non identifying parameters are set; in this case the console log contains the following message for every step: "Step already complete or not restartable, so no action to execute: ...").

## Stopping a Job

The command line option to stop a running execution is as follows:

```
java org.springframework.batch.core.launch.support.CommandLineJobRunner classpath:config/app/batch/beans-billexport.xml -stop billExportJob
```

or

```
java io.oasp.module.batch.common.base.SpringBootBatchCommandLine
io.oasp.gastronomy.restaurant.SpringBootBatchApp classpath:config/app/batch/beans-billexport.xml
billExportJob -stop
```

Note that the job is not shutdown immediately, but might actually take some time to stop.

## Scheduling

In real world scheduling of batches is not as simple as it first might look like.

- Multiple batches have to be executed in order to achieve complex tasks. If one of those batches fails the further execution has to be stopped and operations should be notified for example.
- Input files or those created by batches have to be copied from one node to another.
- Scheduling batch executing could get complex easily (quarterly jobs, run job on first workday of a month, ...)

For Devonfw we propose the batches themselves should not mess around with details of batch administration. Likewise your application should not do so.

Batch administration should be externalized to a dedicated batch administration service or scheduler. This service could be a complex product or a simple tool like cron. We propose [Rundeck](#) as an open source job scheduler.

This gives full control to operations to choose the solution which fits best into existing administration procedures.

## 3.11.2 Implementation

In this chapter we will describe how to properly setup and implement batches.

### Main Challenges

At a first glimpse, implementing batches is much like implementing a backend for client processing. There are, however, some points at which batches have to be implemented totally different. This is especially true if large data volumes are to be processed.

The most important points are:

#### Transaction handling

For processing request made by clients there is usually one transaction for each request. If anything goes wrong, the transaction is rolled back and all changes are reverted.

A naive approach for batches would be to execute a whole batch in one single transaction so that if anything goes wrong, all changes are reverted and the batch could start from scratch. For processing

large amounts of data, this is technically not feasible, because the database system would have to be able to undo every action made within this transaction. And the space for storing the undo information needed for this (the so called "undo tablespace") is usually quite limited.

So there is a need of short running transactions. To help programmers to do so, Spring Batch offers the so called chunk processing which will be explained [here](#).

### Restarting Batches

In client processing mode, when an exception occurs, the transaction is rolled back and there is no need to worry about data inconsistencies.

This is not true for batches however, due to the fact that you usually can't have just one transaction. When an unexpected error occurs and the batch aborts, the system is in a state where the data is partly processed and partly not and there needs to be some sort of plan how to continue from there.

Even if a batch was perfectly reliable, there might be errors that are not under the control of the application, e.g. lost connection to the database, so that there is always a need for being able to restart.

The section on [restarts](#) describes how to design a batch that is restartable. What's important is that a programmer has to invest some time upfront for a batch to be able to restart after aborts.

### Exception handling in Batches

The problem with exception handling is that e.g. a single record can cause a whole batch to fail and many records will remain unprocessed. In contrast to this, in client processing mode when processing fails this usually affects only one user.

To prevent this situation, Spring Batch allows to skip data when certain exceptions occur. However, the feature should not be misused in a way that you just skip all exceptions independently of their cause.

So when implementing a batch, you should think about what exceptional situations might occur and how to deal with that and whether it is okay to skip those exceptions or not. When an unexpected exception occurs, the batch should still fail so that this exception is not ignored but its causes are analyzed.

Another way of handling exceptions in batches is retrying: Simply try to process the data once more and hope that everything works well this time. This approach often works for database problems, e.g. timeouts.

The section on [exception handling](#) explains skipping and retrying in more detail.

Note that exceptions are another reason why you should not execute a whole batch in one transaction. If anything goes wrong, you could either rollback the transaction and start the batch from scratch or you could manually revert all relevant changes. Both are not very good solutions.

### Performance issues

In client processing mode, optimizing throughput (and response times) is an important topic as well, of course.

However, a performance that is still considered okay for client processing might be problematic for batches as these usually have to process large volumes of data and the time for their execution is usually quite limited (batches are often executed at night when no one is using the application).

An example: If processing the data of one person takes a second, this is usually still considered OK for client processing (even though performance could be better). However if a batch has to process the data of 100.000 persons in one night and is not executed with multiple threads, this takes roughly 28 hours, which is by far too much.

The section on [performance](#) contains some tips how to deal with performance problems.

## Setup

### Database

Spring Batch needs some meta data tables for monitoring batch executions and for restoring state for [restarts](#). Detailed description about needed tables, sequences and indexes can be found in [Spring Batch - Reference Documentation: Appendix B. Meta-Data Schema](#).

It is not recommended to add additional meta data tables, because this easily leads to inconsistencies with what is stored in those tables maintained by Spring Batch. You should rather try to extract all needed information out of the standard tables in case the standard API (especially JobRepository and JobExplorer, see below) does not fit your needs.

### Failure information

BATCH\_JOB\_EXECUTION.EXIT\_MESSAGE and BATCH\_STEP\_EXECUTION.EXIT\_MESSAGE store a detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible. BATCH\_STEP\_EXECUTION\_CONTEXT.SHORT\_CONTEXT stores a stringified version of the step's ExecutionContext (see [saving and restoring state](#), the rest is stored in a BLOB if needed). The default length of those columns in the sample schema scripts is 2500.

It is good to increase the length of those columns as far as the database allows it to make it easier to find out which exception failed a batch (not every exception causes a failure, see [exception handling](#)). Some JDBC drivers cast CLOBs to string automatically. If this is the case, you can use CLOBs instead.

### General Configuration

For configuring batches, we recommend not to use annotations (would not work very well for batches) or JavaConfig, but XML, because this makes the whole batch configuration more transparent, as its structure and implementing beans are immediately visible. Moreover the Spring Batch documentation focuses rather on XML based configurations than on JavaConfig.

For explanations on how these XML files are build in general, have a look at the [spring documentation](#).

There is, however, some general configuration needed for all batches, for which we use JavaConfig, as it is also used for the setup of all other layers. You can find an example of such a configuration in the example application (`oasp4j-samples-core` project): BeansBatchConfig. In this section, we will explain the most important parts of this class.

The `jobRepository` is used to update the meta data tables.

The database type can optionally be set on the `jobRepository` for correctly handling database specific things using the `setDatabaseType` method. Possible values are oracle, mysql, postgres, ...

If the size of all three columns, which per default have a length limitation of 2500, has been increased as proposed [here](#), the property `maxVarCharLength` should be adjusted accordingly using the corresponding setter method in order to actually utilize the additional space.

The *jobExplorer* offers methods for reading from the meta data tables in addition to those methods provided by the *jobRepository*, e.g. getting the last executions of a batch.

The *jobLauncher* is used to actually start batches.

We use our own implementation (*JobLauncherWithAdditionalRestartCapabilities*) here, which can be found in the module *modules/batch* (*oasp4j-batch*). It enables a special form of restarting a batch ("restart from scratch", see the section on [restarts](#) for further details).

The *jobRegistry* is basically a map, which contains all batch jobs. It is filled by the bean of type *JobRegistryBeanPostProcessor* automatically.

A *JobParametersIncremeter* (bean "incrementer") can be used to generate unique parameters, see [restarts](#) and [parameters](#) for further details. It should be configured manually for each batch job, see example batch below, otherwise exceptions might occur when starting batches.

### Example-Batch

As already mentioned, every batch job consists of one or more batch steps, which internally either use chunk processing or tasklet based processing.

Our bill export batch job consists of the following to steps:

1. Read all (not processed) bills from the database, mark them as processed (additional attribute) and write them into a CSV file (to be further processed by other systems). This step is implemented using chunk processing (see [chunk processing](#)).
2. Delete all bill from the database which are marked as processed. This step is implemented in a tasklet (see [tasklet based processing](#)).

Note that you could also delete the bills directly. However, for being able to demonstrate tasklet based processing, we have created a separate step here.

Also note that in real systems you would usually create a backup of data as important as bills, which is not done here.

The beans-billexport.xml (located in *src/main/resources/config/app/batch*) has to look like this to implement the batch. Note that you might not fully understand this example by now, but you should after reading the whole chapter on batches.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation=
           "http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/batch
            http://www.springframework.org/schema/batch/spring-batch.xsd">

    <batch:job id="billExportJob" incrementer="incrementer">

        <batch:step id="createCsvFile" next="deleteBills">
            <batch:tasklet>
                <batch:transaction-attributes timeout="180"/>
                <batch:chunk reader="unprocessedBillsReader" processor="processedMarker"
                           writer="csvFileWriter" commit-interval="1000" />
            </batch:tasklet>
            <listeners>
```

```
<listener ref="chunkLoggingListener"/>
<listeners>
</batch:step>

<batch:step id="deleteBills">
<batch:tasklet ref="billsDeleter">
<batch:transaction-attributes timeout="180" />
</batch:tasklet>
</batch:step>

</batch:job>

<bean id="unprocessedBillsReader"
class="io.oasp.salesmanagement.batch.impl.billexport.UnprocessedBillsReader">
<property name="page_size" value="1000" />
<property name="billDao" ref="billDao" />
</bean>

<bean id="processedMarker"
class="io.oasp.salesmanagement.batch.impl.billexport.ProcessedMarker">
<property name="billDao" ref="billDao" />
</bean>

<bean id="csvFileWriter" class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step">
<property name="resource" value="#{jobParameters['outputFile']}" />
<property name="encoding" value="UTF-8" />
<property name="headerCallback">
<bean class="io.oasp.salesmanagement.batch.impl.billexport.BillHeaderCallback" />
</property>
<property name="lineAggregator">
<bean class="io.oasp.salesmanagement.batch.impl.billexport.BillLineAggregator" />
</property>
</bean>

<bean id="billsDeleter" class="io.oasp.salesmanagement.batch.impl.billexport.BillsDeleter">
<property name="billsToDeleteInTransaction" value="10000" />
</bean>

<bean id="chunkLoggingListener"
class="io.oasp.module.batch.common.impl.ChunkLoggingListener" />
</beans>
```

As you can see, there is a job element (billExportJob), which contains the two step elements (createCsvFile and deleteBills). Note that for every step you have to explicitly specify which step comes next (using the next attribute), unless it is the last step.

The step elements always contains a tasklet element, even if chunk processing is used. The transaction-attributes element is especially used to set timeouts of transactions (in seconds). Note that there is usually more than one transaction per step (see below).

What follows is either a chunk element with ItemReader, ItemProcessor, ItemWriter and a commit interval (see [chunk processing](#)) or the tasklet element contains a reference to a tasklet.

In the example above the ItemReader unprocessedBillsReader always reads 1000 ids of unprocessed bills (via a DAO) and returns them one after another. The ItemProcessor processedMarker reads the corresponding bills from the database (see [chunk processing](#) why we do not read them directly in the ItemReader) and marks them as processed. The ItemWriter csvFileWriter (see below on how this writer is configured) writes them to a CSV file. The path of this file is provided as batch parameter ("outputFile").

The tasklet billsDeleter deletes all processed bills (10.000 in one transaction).

The chunkLoggingListener, which is also used in the example above, can be utilized for all chunk steps to log exceptions together with the items where these exceptions occurred (see [listeners](#) for further

details on listeners). Its implementation can be found in the module `modules/batch`. Note that classes used for items have to have an appropriate `toString()` method in order for this listener to be useful.

## Restarts

A batch execution is considered a restart, if it was run already (with the same parameters) and there was a (non skippable) failure or the batch has been stopped.

There are basically two ways how to do a restart:

- Undo all changes and restart from scratch.
- Restore the state of that batch at the time the error occurred and continue processing.

The first approach has two major disadvantages: One is that depending on what the batch does, reverting all of its changes can get quite complex. And you easily end up having implemented a batch that is restartable, but not if it fails in the wrong step.

The second disadvantage is that if a batch runs for several hours and then it fails it has to start all over again. And as the time for executing batches is usually quite limited, this can be problematic.

If reverting all changes is as easy as deleting all files in a given directory or something like that and the expected duration for an execution of the batch is rather short, you might consider the option of always starting at the beginning, otherwise you shouldn't.

Spring Batch supports implementing the second option. Per default, if a batch is restarted with the same parameters as a previous execution of this batch, then this new execution continues processing at the step where the last execution was stopped or failed. If the last execution was already complete, an exception is raised.

The step itself has to be implemented in a way so that it can restore its internal state, which is the main drawback of this second option.

However, there are 'standard implementations' that are capable of doing so and these can easily be adapted to your needs. They are introduced in the section on [chunk processing](#).

For telling Spring Batch to always restart a batch at the very beginning even though there has been an execution of this batch with the same parameters already, set the `restartable` attribute of the `Job` element to `false`.

Per default, setting this attribute to `false` means that the batch is not restartable (i.e. it cannot be started with the same parameters once more). It would raise an error if there was attempt to do so, so that it cannot be restarted where it left off.

We use our own `JobLauncher` (`JobLauncherWithAdditionalRestartCapabilities`) as described in the section on the [general configuration](#) to modify this behavior so that those batches are always restarted from the first step on by adding an extra parameter (instead of raising an exception), so that you do not have to take care of that yourself. So don't think of a batch marked with `restartable="false"` as a batch that is not restartable (as most people would probably assume just looking at the attribute) but as a batch that restarts always from the first step on.

Note that if a batch is restartable by restoring its internal state, it might not work correctly if the batch is started with different parameters after it failed, which usually comes down to the same thing as restating

it from scratch. So the batch has to be restarted and complete successfully before executing the next regular 'run'. When scheduling batches, you should make that sure.

## Chunk Processing

Chunk processing is item based processing. Items can be bills, persons or whatever needs to be processed. Those items are grouped into chunks of a fixed size and all items within such a chunk are processed in one transaction. There is not one transaction for every single (small) item because there would be too many commits which degrades performance.

All items of a chunk are read by an ItemReader (e.g. from a file or from database), processed by an ItemProcessor (e.g. modified or converted) and written out as a whole by an ItemWriter (e.g. to a file or to database).

The size of a chunk is also called commit interval. Careful when choosing a large chunk size: When a skip or retry occurs for a single item (see [exception handling](#)), the current transaction has to be rolled back and all items of the chunk have to be reprocessed. This is especially a problem when skips and retries occur more often and results in long runtimes.

The most important advantages of chunk processing are:

- good trade-off between size and number of transactions (configurable via commit size)
- transaction timeouts that do not have to be adapted for larger amounts of data that needs to be processed (as there is always one transaction for a fixed number of items)
- an exception handling that is more fine-grained than aborting/restarting the whole batch (item based skipping and retrying, see [exception handling](#))
- logging items where exceptions occurred (which makes failure analysis much more easy)

Note that you could actually achieve similar results using [tasklets](#) as described below. However, you would have to write many lines of additional code whereas you get these advantages out of the box using chunk processing (logging exceptions and items where these exceptions occurred is an extension, see [example batch](#)).

Also note that items should not be too "big". For example, one might consider processing all bills within one month as one item. However, doing so you would not have those advantages any more. For instance, you would have larger transactions, as there are usually quite a lot of bills per month or payment method and if an exception occurs, you would not know which bill actually caused the exception. Additionally you would lose control of commit size, since one commit would comprise many bills hard coded and you cannot choose smaller chunks.

Nevertheless, there are sometimes situations where you cannot further "divide" items, e.g. when these are needed for one single call to an external system (e.g. for creating a PDF of all bills within a certain month, if PDFs are created by an external system). In this case you should do as much of the processing as possible on the basis of "small" items and then add an extra step to do what cannot be done based on these "small" items.

### ItemReader

A reader has to implement the ItemReader interface, which has the following method:

```
public T read() throws Exception;
```

T is a type parameter of the ItemReader interface to be replaced with the type of items to be read.

The method returns all items (one at a time) that need to be processed or null if there are no more items.

If an exception occurs during read, Spring Batch cannot tell with item caused the exception (as it has not been read yet). That is why a reader should contain as little processing logic as possible, minimizing the potential for failures.

## Caching

Per default, all items read by an ItemReader are cached by Spring Batch. This is useful because when a skippable exception occurs during processing of a chunk, all items (or at least those, that did not cause the exception) have to be reprocessed. These items are not read twice but taken from the cache then.

This is often necessary, because if a reader saves its current state in member variables (e.g. the current position within a list of items) or uses some sort of cursor, these will be updated already and the next calls of the read method would deliver the next items already and not those that have to be reprocessed.

However this also means that when the items read by an ItemReader are entities, these might be detached, because these might have been read in a different transaction. In some standard implementations Spring Batch even manually detaches entities in ItemReaders.

In case these entities are to be modified it is a good practice that the ItemReader only reads IDs and the ItemProcessor loads the entities for these IDs to avoid the problem.

## Reading from Transactional Queues

In case the reader reads from a transactional queue (e.g. using JMS), you must not use caching, because then an item might get processed twice: Once from cache and once from queue to where it has been returned after the rollback. To achieve this, set reader-transactional-queue="true" in the chunk element in the step definition.

Moreover the equals and hashCode methods of the class used for items have to be appropriately implemented for Spring Batch to be able to identify items that were processed before unsuccessfully (causing a rollback and thereby returning them to the queue). Otherwise the batch might be caught in an infinite loop trying to process the same item over and over again (e.g. when the item is about to be skipped, see [exception handling](#)).

## Reading from the Database

When selecting data from a database, there is usually some sort of cursor used. One challenge is to make this cursor not participate in the chunk's transaction, because it would be closed after the first chunk.

We will show how to use JDBC based cursors for ItemReader's in later releases of this documentation.

For JPA/JPQL based queries, cursors cannot be used, because JPA does not know of the concept of a cursor. Instead it supports pagination as introduced in the chapter on the data access layer, which can be used for this purpose as well. Note that pagination requires the result set to be sorted in an unambiguous order to work reliably. The order itself is irrelevant as long it does not change (you can e.g. sort the entities by their primary key).

ItemReader's using pagination should inherit from the AbstractPagingItemReader, which already provides most of the needed functionality. It manages the internal state, i.e. the current position, which can be correctly restored after a restart (when using an unambiguous order for the result set).

Classes inheriting from `AbstractPagingItemReader` must implement two methods.

The method `doReadPage()` performs the actual read of a page. The result is not returned (return type is `void`) but used to replace the content of the 'results' instance variable (type: `List`).

Due to our layering concept and the persistence layer being the only place where accesses to the database should take place, you should not directly execute a query in this method, but call a DAO, which itself executes the query (using pagination).

`AbstractPagingItemReader` provides methods for finding out the current position: use `getPage()` for the current page and `getPageSize()` for the (max.) page size. These values should be passed to the DAO as parameters. Note that the `AbstractPagingItemReader` starts counting pages from zero, whereas the `PaginationTo` used for pagination (retrieved by calling `SearchCriteriaTo.getPagination()`) starts counting from one, which is why you always have to increment the page number by one.

The second method is `doJumpToPage(int)`, which usually only requires an empty implementation.

Furthermore, you need to set the property `pageSize`, which specifies how many items should be read at once. A page size that is as big as the commit interval usually results in the best performance.

The approach of using pagination for `ItemReader`'s should not be used when items (usually entities) are added or removed or modified by the batch step itself or in parallel with the execution of the batch step so that the order changes, e.g. by other batches or due to operations started by clients (i.e. if the batch is executed in online mode). In this case there might be items processed twice or not processed at all. Be aware that due to hibernates Hi/Lo-Algorithm newer entities could get lower IDs than existing IDs and you probably will not process all entities if you rely on strict ID monotony!

A simple solution for such scenarios would be to introduce a new flag 'processed' for the entities read if that is an option (as it is also done in the example batch). The query should be rewritten then so that only unprocessed items are read (additionally limiting the result set size to the number of items to be processed in the current chunk, but not more).

Note that most of the standard implementations provided by Spring Batch do not fit to the layering approach in OASP4J applications, as these mostly require direct access to an `EntityManager` or a `JDBC` connection for example. You should think twice when using them and break the layering concept.

## Reading from Files

For reading simply structured files, e.g. for those in which every line corresponds to an item to be processed by the batch, the `FlatFileItemReader` can be used. It requires two properties to be set: The first one the `LineMapper` (property `lineMapper`), which is used to convert a line (i.e. a `String`) to an item. It is a very simple interface which will not be discussed in more detail here. The second one is the `resource`, which is actually the file to be read. When set in the XML, it is sufficient to specify the path with a "file:" in front of it if it is a normal file from the file system.

In addition to that, the property `linesToSkip` (integer) can be set to skip headers for example. For reading more than one line before for creating an item a `RecordSeparatorPolicy` can be used, which will not be discussed in more detail here, too. Per default, all lines starting with a '#' will be considered to be a comment, which can be changed by changing the `comment` property (string array). The `encoding` property can be used to set the encoding. A `FlatFileItemReader` can restore its state after restarts.

For reading XML files, you can use the `StaxEventItemReader` (`StAX` is an alternative to `DOM` and `SAX`), which will not be discussed in further detail here.

In case the standard implementations introduced here do not fit your needs, you will need to implement your own ItemReader. If this ItemReader has some internal state (usually stored in member variables), which needs to be restored in case of restarts, see the section on [saving and restoring state](#) for information on how to do this.

### ItemProcessor

A processor must implement the ItemProcessor interface, which has the following method:

```
public O process(I item) throws Exception;
```

As you can see, there are two type parameters involved: one for the type of items received from the ItemReader and one for the type of items passed to the ItemWriter. These can be the same.

If an item has been selected by the ItemReader, but there is no need to further process this item (i.e. it should not be passed to the ItemWriter), the ItemProcessor can return null instead of an item.

Strictly interpreting chunk processing, the ItemProcessor should not modify anything but should only give instructions to the ItemWriter how to do modifications. For entities however this is not really practical and as it requires no special logic in case of rollbacks/restarts (as all modifications are transactional), it is usually OK to modify them directly.

In contrast to this, performing accesses to files or calling external systems should only be done in ItemReader's/ItemWriter's and the code needed for properly handling failures (restarts for example) should be encapsulated there.

It is usually a good practice to make ItemProcessor's stateless, as the process method might be called more than once for one item (see the section on ItemReader's why). If your ItemProcessor really needs to have some internal state, see [saving and restoring state](#) on how to save and restore the state for restarts.

Do not forget to implement use cases instead of implementing everything directly in the ItemProcessor if the processing logic gets more complex.

### ItemWriter

A writer has to implement the ItemWriter interface, which has the following method:

```
public void write(List<? extends T> items) Exception;
```

This method is called at the end of each chunk with a list of all (processed) items. It is not called once for every item, because it is often more efficient doing 'bulk writes', e.g. when writing to files.

Note that this method might also be called more than once for one item (see the section on ItemReader's why).

At the end of the write method, there should always be a flush.

When writing to files, this should be obvious, because when a chunk completes, it is expected that all changes are already there in case of restarts, which is not true if these changes were only buffered but have not been written out.

When modifying the database, the flush method on the EntityManager should be called, too (via a DAO), because otherwise there might be changes not written out yet and therefore constraints were

not checked yet. This can be problematic, because Spring Batch considers all exceptions that occur during commit as critical, which is why these exceptions cannot be skipped. You should be careful using deferred constraints for the same reason.

## Writing to Database or Transactional Queues

All changes made which are transactional can be conducted directly, there is no special logic needed for restarts, because these changes are applied if and only if the chunk succeeds.

## Writing to Files

For writing simply structured files, the `FlatFileItemWriter` can be used. Similar to the `FlatFileItemReader` it requires the resource (i.e. the file) and a `LineAggregator` (property `lineAggregator`; instead of the `lineMapper`) to be set.

There are various properties that can be used of which we will only present the most important ones here. As with the `FlatFileItemReader`, the `encoding` property is used to set the encoding. A `FlatFileHeaderCallback` (property `headerCallback`) can be used to write a header.

The `FlatFileItemWriter` can restore its state correctly after restarts. In case the files contains too many line (written out in chunks that did not complete successfully), these lines are removed before continuing execution.

For writing XML files, you can use the `StaxEventItemWriter`, which will not be discussed in further detail here.

Just as with `ItemReader`'s and `ItemProcessor`'s: In case your `ItemWriter` has some internal state this state is not managed by a standard implementation, see [saving and restoring state](#) on how to make your implementation restartable (restart by restoring the internal state).

## Saving and Restoring State

For saving and restoring (in case of restarts) state, e.g. saving and restoring values of member variables, the `ItemStream` interface should be implemented by the `ItemReader`/`ItemProcessor`/`ItemWriter`, which has the following methods:

```
public void open(ExecutionContext executionContext) throws ItemStreamException;
public void update(ExecutionContext executionContext) throws ItemStreamException;
public void close() throws ItemStreamException;
```

The `open` method is always called before the actual processing starts for the current step and can be used to restore state when restarting.

The `ExecutionContext` passed in as parameter is basically a map to be used to retrieve values set before the failure. The method `containsKey(String)` can be used to check if a value for a given key is set. If it is not set, this might be because the current batch execution is no restart or no value has been set before the failure.

There are several getter methods for actually retrieving a value for a given key: `get(String)` for objects (must be serializable), `getInt(String)`, `getLong(String)`, `getDouble(String)` and `getString(String)`. These values will be the same as after the subsequent call to the `update` method after the last chunk that completed successfully. Note that if you update the `ExecutionContext` outside of the `update` method (e.g. in the `read` method of an `ItemReader`), it might contain values set in chunks that did not finish successfully after restarts, which is why you should not do that.

So the update method is the right place to update the current state. It is called after each chunk (and before and after each step).

For setting values, there are several put methods: put(String, Object), putInt(String, int), putLong(String, long), putDouble(String, double) and putString(String, String). You can choose keys (String) freely as long as these are unique within the current step.

Note that when a skip occurs, the update method is sometimes but not always called, so you should design your code in a way that it can deal with both situations.

The close method is usually not needed.

Do not misuse the ItemStream interface for purposes other than storing/restoring state. For instance, do not use the update method for flushing, because you will not have the chance to properly handle failure (e.g. skipping). For opening or closing a file handle, you should rather use a StepExecutionListener as introduced in the section on [listeners](#). The state can also be restored in the beforeStep(ExecutionListener) method (instead of the open method).

Note that when a batch that always starts from scratch (i.e. the restartable attribute has been set to false for the batch job) is restarted, the ExecutionContext will not contain any state from the previous (failed) execution, so there is no use in storing the state in this case and usually no need to, of course, because the batch will start all over again.

## Tasklet based Processing

Tasklets are the alternative to chunk processing. In the section on [chunk processing](#) we already mentioned the advantages of chunk processing as compared to tasklets. However, if only very few data needs to be processed (within one transaction) or if you need to do some sort of bulk operation (e.g. deleting all records from a database table), where the currently processed item does not matter and it is unlikely that a 'fine grained' exception handling will be needed, tasklets might still be considered an option. Note that for the latter use case you should still use more than one transaction, which is possible when using tasklets, too.

Tasklets have to implement the interface with the same name, which has the following method:

```
public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception;
```

This method might be called several times. Every call is executed inside a new transaction automatically. If processing is not finished yet and the execute method should be called once more, just use RepeatStatus.CONTINUABLE as return value and RepeatStatus.FINISHED otherwise.

The StepContribution parameter can be used to set how many items have been processed manually (which is done automatically using chunk processing), there is, however, usually no need to do so.

The ChunkContext is similar to the ExecutionContext, but is only used within one chunk. If there is a retry in chunk processing, the same context should be used (with the same state that this context had when the exception occurred).

Note that tasklets serve as the basis for chunk processing internally. For chunk processing there is a Spring Batch internal tasklet, which has an execute method that is called for every chunk and itself calls ItemReader, ItemProcessor and ItemWriter.

That is the reason why a StepContribution and a ChunkContext are passed to tasklets as parameters, even though they are more useful in chunk processing. Moreover this is also the reason why you have to

use the tasklet element in the XML even though you want to specify a step that uses chunk processing (see [the example batch](#)).

## Exception Handling

As already mentioned, in chunk processing you can configure a step so that items are skipped or retried when certain exceptions occur.

If retries are exhausted (per default, there is no retry) and the exception that occurred cannot be skipped (per default, no exception can be skipped), the batch will fail (i.e. stop executing).

In tasklet based processing this cannot be done, the only chance is to implement the needed logic yourself.

## Skippping

Before skipping items you should think about what to do if a skip occurs. If a skip occurs, the exception will be logged in the server log. However if no one evaluates those logs on a regular basis and informs those who are affected further actions need to take place when implementing the batch.

Implement the SkipListener interface to be informed when a skip occurs. For example, you could store a notification or send a message to someone. For skips that occurred in ItemReader's there is no information available about the item that was skipped (as it has not been read yet) which is why there should be as little processing logic as possible in an ItemReader. It might also be a reason why you might want to forbid to skip exceptions that might occur in readers.

Do not try to catch skipped exceptions and write something into the database in a new transaction (e.g. a notification) instead of using a SkipListener, because a skipped item might be processed more than once before actually being skipped (for example, if a skippable exception is thrown during a call of an ItemWriter, Spring Batch does not know which item of the current chunk actually caused the exception and therefore has to retry each item separately in order to know which item actually caused the exception).

Skippable exception classes can be specified as shown below:

```
<batch:chunk ... skip-limit="10">
  <batch:skippable-exception-classes>
    <batch:include class="..."/>
    <batch:include class="..."/>
    ...
  </batch:skippable-exception-classes>
</batch:chunk>
```

The attribute skip-limit, which has to be set in case there is any skippable exception class configured, is used to set how many items should be skipped at most. It is useful to avoid situations where very many items are skipped but the batch still completes successfully and no one notices this situation.

Skippable exception classes are specified by their fully qualified name (e.g. java.lang.Exception), each of such class set in its own include element as shown above. Subclasses of such classes are also skipped.

To programmatically decide whether to skip an exception or not, you can set a skip policy as shown below:

```
<batch:chunk ... skip-policy="mySkipPolicy">
```

The skip policy (here `mySkipPolicy`) has to be a bean that implements the interface `SkipPolicy` with the following method:

```
public boolean shouldSkip(java.lang.Throwable t,  
                           int skipCount)  
                           throws SkipLimitExceededException
```

To skip the exception and continue processing, just return true and otherwise false.

The parameter `skipCount` can be used for a skip limit. A `SkipLimitExceededException` should be thrown if there should be thrown if there should be no more skips. Note that this method is sometimes called with a `skipCount` less than zero to test if an exception is skippable in general.

When a `SkipPolicy` is set, the attribute `skip-limit` and element `skippable-exception-classes` are ignored.

You could of course skip every exception (using `java.lang.Exception` as skippable exception class). This is, however, not a good practice as it might easily result in an error in the code that is ignored as the batch still completes successfully and everything seems to be fine. Instead, you should think about what kind of exceptions might actually occur, what to do if they occur and if it is OK to skip them. If an unexpected exception occurs, it is usually better to fail the batch execution and analyze the cause of the exception before restarting the batch.

Exceptions that can occur in `ItemWriter`'s that write something to file should not be skipped unless the `ItemWriter` can properly deal with that. Otherwise there might be data written out even though the according item is skipped, because operations in the file systems are not transactional.

Another situation where skips can be problematic is when calls to external interfaces are being made and these calls change something "on the other side", as these calls are usually not transactional. So be careful using skips here, too.

### Retrying

For some types of exceptions, processing should be retried independently of whether the exception can be skipped or would otherwise fail the batch execution.

For example, if there was a database timeout, this might be because there were too many requests at the time the chunk was processed. And it is not unlikely that retrying to successfully complete the chunk would succeed.

There are, of course, also exceptions where retrying does not make much sense. E.g. exceptions caused by the business logic should be deterministic and therefore retrying does not make much sense in this case.

Nevertheless, retrying every exception results in longer runtime but should in general be considered OK if you do not know which exceptions might occur or do not have the time to think about it.

Retryable exception classes can be set similarly to setting skippable exception classes:

```
<batch:chunk ... retry-limit="3">  
  <batch:retryable-exception-classes>  
    <batch:include class="..."/>  
    <batch:include class="..."/>  
    ...  
  </batch:retryable-exception-classes>  
</batch:chunk>
```

The retry-limit attribute specifies how many times one individual item can be retried, as long as the exception thrown is "retryable".

As with skippable exception classes, retryable exception classes are set in include elements and their subclasses are retried, too.

To programmatically decide, whether to retry an exception or not, you can use a `RetryPolicy`, which is not covered in more detail here.

Note that even if no retry is configured, an item might nevertheless be processed more than once. This is because if a skippable exception occurs in a chunk, all items of the chunk that did not cause the exception have to be reprocessed, which is done in a separate transaction for every item, as the transaction in which these items were processed in the first place was rolled back. And even if the exception is not skippable, there is no guarantee that Spring Batch will not attempt to reprocess each item separately.

## Listeners

Spring Batch provides various listeners for various events to be notified about.

For every listener there is an interface which can either be implemented by an `ItemReader`, `ItemProcessor`, `ItemWriter` or `Tasklet` or by a separate listener class, which can be registered for a step like this:

```
<batch:tasklet>
  <batch:chunk .../>
  <batch:listeners>
    <batch:listener ref="listener1"/>
    <batch:listener ref="listener2"/>
    ...
  </batch:listeners>
</batch:tasklet>
<beans:bean id="listener1" class="..."/>
<beans:bean id="listener2" class="..."/>
...
```

The most commonly used listener is probably the `StepExecutionListener`, which has methods that are called before and after the execution of the step. It can be utilized e.g. for opening and closing files.

The following example shows how to use the listener:

```
public class MyListener implements StepExecutionListener {

    public void beforeStep(StepExecution stepExecution) {
        // take actions before processing of the step starts
    }

    public ExitStatus afterStep(StepExecution stepExecution) {
        try {
            // take actions after processing is finished
        } catch (Exception e) {
            stepExecution.addFailureException(e);
            stepExecution.setStatus(BatchStatus.FAILED);
            return ExitStatus.FAILED.addExitDescription(e);
        }
        return null;
    }
}
```

In the `afterStep(StepExecution)` method, you can check the outcome of the batch execution (completed, failed, stopped etc.) checking the `ExitStatus`, which can be accessed via `StepExecution#getExitStatus()`.

You can even modify the ExitStatus by returning a new ExitStatus, which is something we will not discuss in further detail here. If you do not want to modify the ExitStatus, just return null.

Throwing an exception in this method has no effect. If you want to fail the whole batch in case an exception occurs, you have to do an exception handling as shown above. This does not apply to the beforeStep method.

For other types of listeners (among others the SkipListener mentioned already) see [Spring Batch Reference Documentation - 5. Configuring a Step - Intercepting Step Execution](#).

Note that exception handling for listeners is often a problem, because exceptions are mostly ignored, which is not always documented very well. If an important part of a batch is implemented in listener methods, you should always test what happens when exceptions occur. Or you might think about not implementing important things in listeners ...

If you want an exception to fail the whole batch, you can always wrap it in a FatalStepExecutionException, which will stop the execution.

## Parameters

The section on [starting and stopping batches](#) already showed how to start a batch with parameters.

One way to get access to the values set is using the StepExecutionListener introduced in the section on [listeners](#) like this:

```
public void beforeStep(StepExecution stepExecution) {  
  
    String parameterValue = stepExecution.getJobExecution().getJobParameters().  
        getString("parameterKey");  
}
```

There are getter methods for strings, doubles, longs and dates. Note that when set via the CommandLineJobRunner or SpringBootBatchCommandLine, all parameters will be of type string unless the type is specified in brackets after the parameter key, e.g. processUntil(date)=2015/12/31. The parameter key here is "processUntil".

Another way is to inject values. In order for this to work, the bean has to have step scope, which means there is a new object created for every execution of a batch step. It works like this:

```
<bean id="myProcessor" class="...MyItemProcessor" scope="step">  
    <property name="parameter" value="#{jobParameters['parameterKey']}" />  
</bean>
```

There has to be an appropriate setter method for the parameter of course.

As already mentioned in the section on [restarts](#), a batch that successfully completed with a certain set of parameters cannot be started once more with the same parameters as this would be considered a restart, which is not necessary, because the batch was already finished.

So using no parameters for a batch would mean that it can be started until it completes successfully once, which usually does not make much sense.

As batches are usually not executed more than once a day, we purpose introducing a general "date" parameter (without time) for all batch executions.

It is advisable to add the date parameter automatically in the JobLauncher if it has not been set manually, which can be done as shown below:

```
private static final String DATE_PARAMETER = "date";  
  
...  
  
if (jobParameters.getDate("DATE_PARAMETER") == null) {  
  
    Date dateWithoutTime = new Date();  
    Calendar cal = Calendar.getInstance();  
    cal.setTime(dateWithoutTime);  
    cal.set(Calendar.HOUR_OF_DAY, 0);  
    cal.set(Calendar.MINUTE, 0);  
    cal.set(Calendar.SECOND, 0);  
    cal.set(Calendar.MILLISECOND, 0);  
    dateWithoutTime = cal.getTime();  
  
    jobParameters = new JobParametersBuilder(jobParameters).addDate(  
        DATE_PARAMETER, dateWithoutTime).toJobParameters();  
  
    ... // using the jobParametersIncrementer as shown above  
}
```

Keep in mind that you might need to set the date parameter explicitly for restarts. Also note that automatically setting the date parameter can be problematic if a batch is sometimes started before and sometimes after midnight, which might result in a batch not being executed (as it has already been executed with the same parameters), so at least for productive systems you should always set it explicitly.

The date parameters can also be useful for controlling the business logic, e.g. a batch can process all data that was created until the current date (as set in the date parameter), thereby giving a chance to control how much is actually processed.

If your batch has to run more than once a day you could easily adapt the concept for timestamps. If you are using an external batch scheduler, they often provide a counter for the execution and you might automatically pass this instead of the date parameter.

## Performance Tuning

Most important for performance are of course the algorithms that you write and how fast (and scalable) these are, which is the same as for client processing. Apart from that, the performance of batches is usually closely related to the performance of the database system.

If you are retrieving information from the database, you can have one complex query executed in the ItemReader (via a DAO) retrieving all the information needed for the current set of items, or you can execute further queries in the ItemProcessor (or ItemWriter) on a per item basis to retrieve further information.

The first approach is usually by far more performant, because there is an overhead for every query being executed and this approach results in less queries being executed. Note that there is a tradeoff between performance and maintainability here. If you put everything into the query executed by an ItemReader, this query can get quite complex.

Using cursors instead of pagination as described in the section on [ItemReaders](#) can result in a better performance for the same reason: When using a cursor, the query is only executed once, when using pagination, the query is usually executed once per chunk. You could of course manually cache items, however this easily leads to a high memory consumption.

Further possibilities for optimizations are query (plan) optimization and adding missing database indexes.

## Testing

This section covers how to unit and integration test in detail. Therefore we focus here on testing batches.

In order for the unit test to run a batch job the unit test class must extends the `AbstractSpringBatchIntegrationTest` class. Two annotations are used to load the job's `ApplicationContext`:

`@RunWith(SpringJUnit4ClassRunner.class)`: Indicates that the class should use Spring's JUnit facilities

`@SpringApplicationConfiguration(classes = {...}, locations = {...})`: Indicates which JavaConfig classes (attribute 'classes') and/or XML files (attribute 'locations') contain the `ApplicationContext`. Use `@ContextConfiguration(...)` if Spring Boot is not used.

```
@RunWith(SpringJUnit4ClassRunner.class)
@DirtiesContext(classMode = ClassMode.AFTER_CLASS)
@ActiveProfiles("db-plain")
public abstract class AbstractSpringBatchIntegrationTest {...}
```

```
@SpringApplicationConfiguration(classes= { SpringBootBatchApp.class }, locations = { "classpath:config/app/batch/beans-productimport.xml" })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {...}
```

### Testing Batch Jobs

For testing the complete run of a batch job from beginning to end involves following steps:

Set up a test condition,

execute the job,

Verify the end result.

The test method below begins by setting up the database with test data. The test then launches the Job using the `launchJob()` method. The `launchJob()` method is provided by the `JobLauncherTestUtils` class.

Also provided by the `utils` class is `launchJob(JobParameters)`, which allows the test to give particular parameters. The `launchJob()` method returns the `JobExecution` object which is useful for asserting particular information about the Job run. In the case below, the test verifies that the Job ended with `ExitStatus "COMPLETED"`.

```
@SpringApplicationConfiguration(classes= { SpringBootBatchApp.class }, locations = { "classpath:config/app/batch/beans-productimport.xml" })
public class ProductImportJobTest extends AbstractSpringBatchIntegrationTest {

    @Inject
    private Job billExportJob;

    @Test
    public void shouldExportBills() throws Exception {
        JobExecution jobExecution = getJobLauncherTestUtils(this.billExportJob).launchJob();
        assertThat(jobExecution.getExitStatus()).isEqualTo(ExitStatus.COMPLETED);
    }
}
```

Note that when using the `launchJob()` method, the batch execution will never be considered as a restart (i.e. it will always start from scratch). This is achieved by adding a unique (random) parameter.

This is not true for the method `launchJob(JobParameters)` however, which will result in an exception if the test is executed twice or a batch is executed in two different tests with the same parameters.

We will add methods for appropriately handling this situation in future releases of Devonfw. Until then you can help yourself by using the method `getUniqueJobParameters()` and then add all required parameters to those parameters returned by the method (as shown in the section on [parameters](#)).

Also note that even if skips occurred, the `ExitStatus` is still `COMPLETED`. That is one reason why you should always check whether the batch did what it was supposed to do or not.

## Testing Individual Steps

For complex batch jobs individual steps can be tested. For example to test a `createCsvFile`, run just that particular Step. This approach allows for more targeted tests by allowing the test to set up data for just that step and to validate its results directly.

```
JobExecution jobExecution = getJobLauncherTestUtils(this.billExportJob).launchStep("createCsvFile");
```

## Validating Output Files

When a batch job writes to the database, it is easy to query the database to verify the output. To facilitate the verification of output files Spring Batch provides the class `AssertFile`. The method `assertFileEquals` takes two `File` objects and asserts, line by line, that the two files have the same content. Therefore, it is possible to create a file with the expected output and to compare it to the actual result:

```
private static final String EXPECTED_FILE = "classpath:expected.csv";
private static final String OUTPUT_FILE = "file:./temp/output.csv";
AssertFile.assertFileEquals(new FileSystemResource(EXPECTED_FILE), new FileSystemResource(OUTPUT_FILE));
```

## Testing Restarts

Simulating an exception at an arbitrary method in the code can be done relatively easy using [AspectJ](#). Afterwards you should restart the batch and check if the outcome is still correct.

Note that when using the `launchJob()` method, the batch is always started from the beginning (as already mentioned). Use the `launchJob(JobParameters)` instead with the same parameters for the initial (failing) execution and for the restart.

Test your code thoroughly. There should be at least one restart test for every step of the batch job.

## 3.12 Integrating Spring Data in OASP4J

### 3.12.1 Introduction

This chapter specifies a possible solution for integrating the Spring Data module in devonfw. It is possible to have some services using the Spring Data module while some services use JPA/Hibernate.

To integrate Spring Data in devonfw, there can be two approaches as stated below:

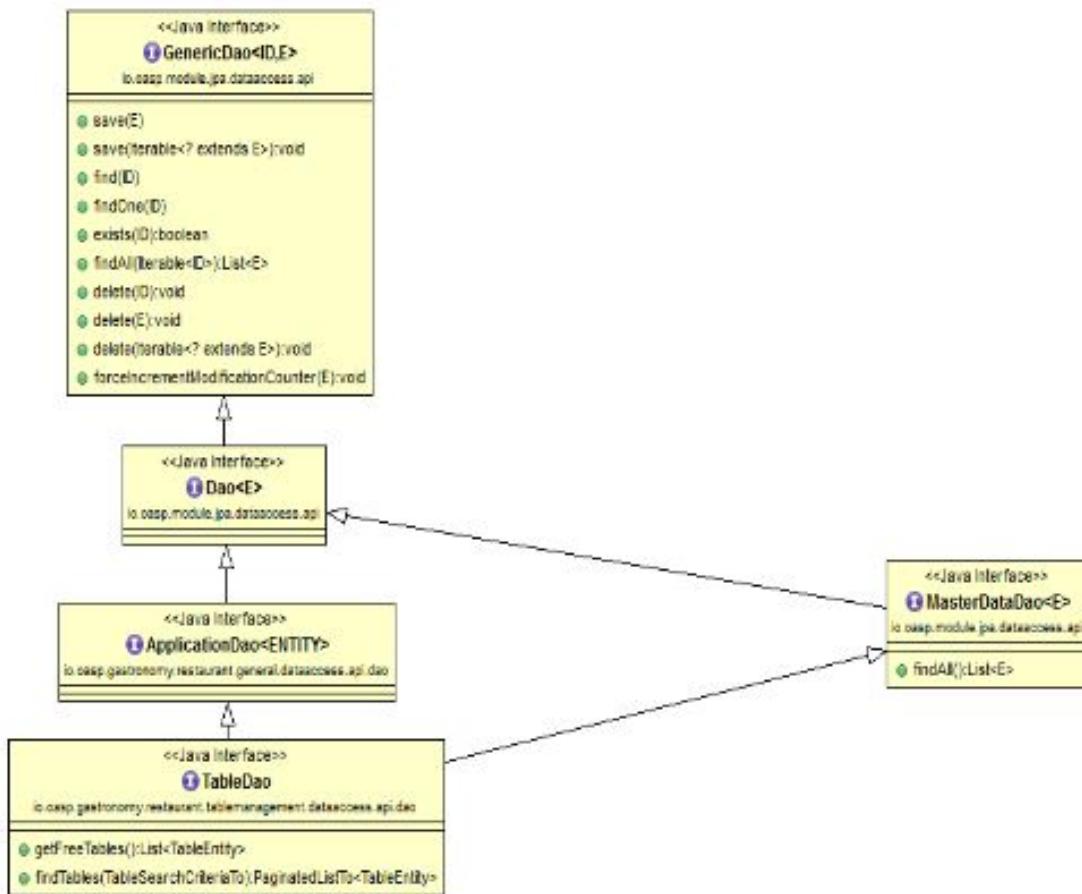
1. Create a new module for spring data
2. Integrate Spring Data in an existing core project

The more feasible approach will be option 2 i.e to integrate it into an existing core project instead of creating a new module. Below are the reasons for not preferring a creation of a different module supporting spring data:

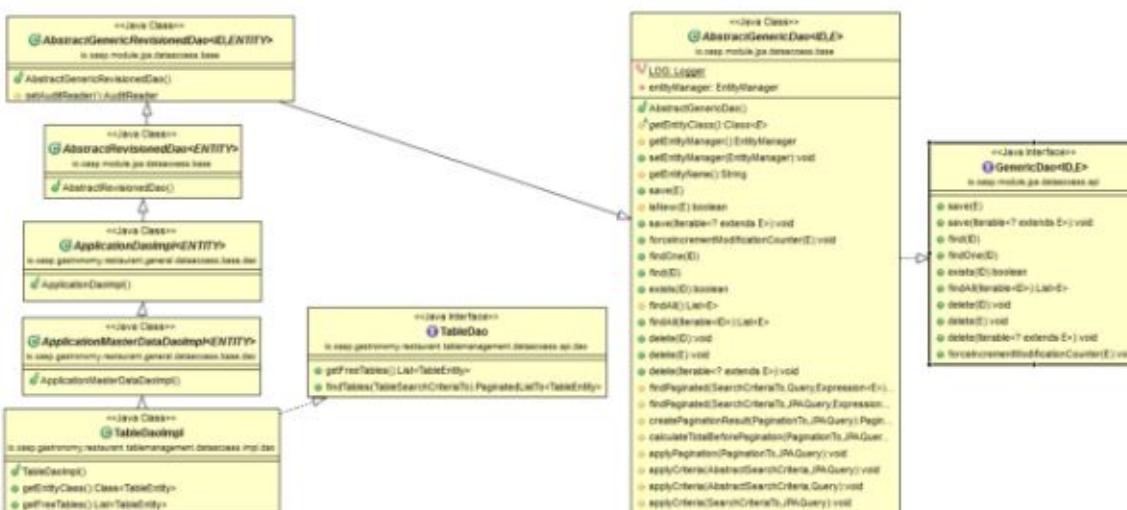
- It does not follow KISS (Keep it simple, stupid) principle. In the existing structure of sample application, Entity classes along with some abstract implementation classes are included in oasp4j-samples-core project. You need to refer these classes while implementing spring-data. If you try to refer it in different module, it will become complex to compare it to the first approach.
- If you integrate Spring Data in oasp4j, you need to annotate `SpringBootApp.java` class with `@Enablejparepositories`. If you create a different module, it will not be possible, as `SpringBootApp` class is in the core module.

[[cookbook-spring-data\_existing-structure of data access layer]] == Existing Structure of Data Access Layer

Consider `TableEntity` as an example.

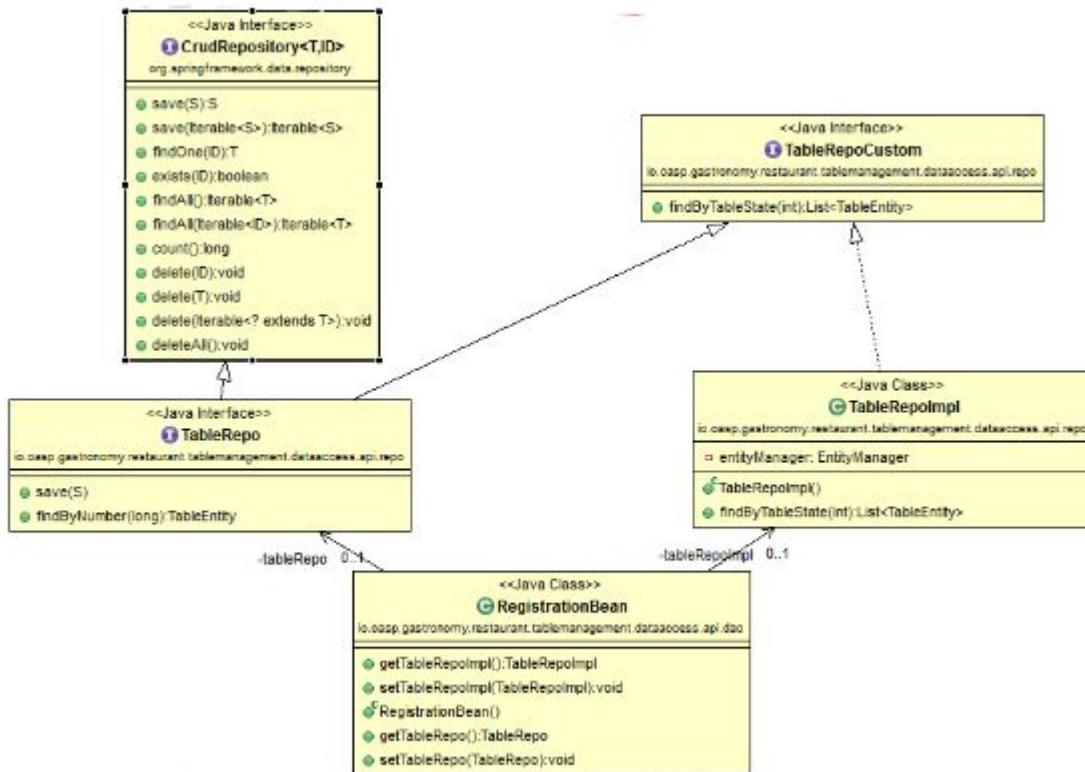


## Dig 1: TableDao structure in oasp4j-sample-core



## Dig 2: Existing Structure of TableDaoImpl

## [[cookbook-spring-data\_structure-of data access layer with spring data]] == Structure of Data Access Layer with Spring Data



**Dig 3: Structure after integrating Spring Data**

Below are the steps, to integrate Spring Data in the data access layer:

- Add dependency for Spring Data in pom.xml of oasp4j-samples-core project

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
  
```

- Create Spring data Repository - Create interface which extends spring data repositories such as CRUDRepository or PagingAndSortingRepository and annotate it with @Repository annotation. Spring data have repositories such as CRUDRepository which provide the default CRUD functionality.

```

@Repository
public interface TableRepo extends CrudRepository<TableEntity, Serializable> {
}
  
```

- Create the class, annotate it with @Component annotation and autowire spring data repository created above.

```

@Component
public class RegistrationBean {
    @Inject
    private TableRepo tableRepo;
    /**
     * The constructor.
     */
    public RegistrationBean() {
    }
}
  
```

```
/***
 * @return tableRepo
 */
public TableRepo getTableRepo() {

    return this.tableRepo;
}

/***
 * @param tableRepo the tableRepo to set
 */
public void setTableRepo(TableRepo tableRepo) {

    this.tableRepo = tableRepo;
}

}
```

- Here, you are ready to test the functionality. Create a test class to test above changes.

```
@SpringApplicationConfiguration(classes = { SpringBootApp.class })
@WebAppConfiguration
@EnableJpaRepositories(basePackages =
{ "io.oasp.gastronomy.restaurant.tablemanagement.dataaccess.api.repo" })
@ComponentScan(basePackages = { "io.oasp.gastronomy.restaurant.tablemanagement.dataaccess.api.dao" })
public class TestClass extends ComponentTest {

    @Inject
    RegistrationBean registrationBean;

    /**
     * @return registrationBean
     */
    public RegistrationBean getRegistrationBean() {

        return this.registrationBean;
    }

    /**
     * @param registrationBean the registrationBean to set
     */

    public void setRegistrationBean(RegistrationBean registrationBean) {

        this.registrationBean = registrationBean;
    }

    /**
     * @param args
     */
    @Test
    public void saveTable() {

        TableEntity table = new TableEntity();
        table.setId(106L);
        table.setModificationCounter(1);
        table.setNumber(6L);
        table.setState(TableState.FREE);
        table.setWaiterId(2L);
    }
}
```

```
        System.out
        .println("TableRepo instance *****" + +
getRegistrationBean());
        TableEntity entity = getRegistrationBean().getTableRepo().save(table);
        System.out.println("entity id " + entity);
    }
}
```

Note: If you get DataIntegrityViolationExceptions while saving an object in a database, modify the script to auto\_increment column id. The database should be able to auto increment column id as you have @GeneratedValue annotation in ApplicationPersistenceEntity.

- Modify SpringBootApp.java class to scan the JPA repositories.

```
@SpringBootApplication(exclude = { EndpointAutoConfiguration.class })
@EntityScan(basePackages = { "io.oasp.gastronomy.restaurant" }, basePackageClasses =
{ AdvancedRevisionEntity.class })
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SpringBootApp {

    /**
     * Entry point for spring-boot based app
     *
     * @param args - arguments
     */
    public static void main(String[] args) {

        SpringApplication.run(SpringBootApp.class, args);
    }
}
```

The above example shows how you can implement default functionalities. If you want to add custom functionalities, then you need to add custom repository and provide its implementation class. Also, you need to modify TableRepo to extend the custom repository. Below are the steps. Make note that, this is in continuation with previous example:

Add custom repository as below in a repo package itself:

```
public interface TableRepoCustom {

    /**
     * @param number
     * @return
     */
    List<TableEntity> findByTableState(int number);
}
```

- Create an implementation class for the above custom repository in a repo package itself. You have not annotated repository with any annotation, still Spring data will consider it as a custom repository. This is because spring data scan the repository package to search for any class and if it found one, then spring data consider it as a custom repository.

```
public class TableRepoImpl implements TableRepoCustom {
    @PersistenceContext
    private EntityManager entityManager;
    /**
     * {@inheritDoc}
     */
    @Override
    public List<TableEntity> findByTableState(int state) {
```

```
String query = "select table from TableEntity table where table.state= " + state;
System.out.println("Query " + query);
List<TableEntity> tableList = this.entityManager.createQuery(query).getResultList();
return tableList;
}
}
```

- Modify test class to include above functionality

```
@SpringApplicationConfiguration(classes = { SpringBootApp.class })
@WebAppConfiguration
@EnableJpaRepositories(basePackages =
{ "io.oasp.gastronomy.restaurant.tablemanagement.dataaccess.api.repo" })
@ComponentScan(basePackages = { "io.oasp.gastronomy.restaurant.tablemanagement.dataaccess.api.dao" })
public class TestClass extends ComponentTest {
    @Inject
    RegistrationBean registrationBean;
    /**
     * @return registrationBean
     */
    public RegistrationBean getRegistrationBean() {
        return this.registrationBean;
    }
    /**
     * @param registrationBean the registrationBean to set
     */
    public void setRegistrationBean(RegistrationBean registrationBean) {
        this.registrationBean = registrationBean;
    }
    /**
     * @param args
     */
    @Test
    public void saveTable() {
        TableEntity table = new TableEntity();
        table.setId(106L);
        table.setModificationCounter(1);
        table.setNumber(6L);
        table.setState(TableState.FREE);
        table.setWaiterId(2L);
        System.out
            .println("TableRepo instance ***** " +
getRegistrationBean());
        TableEntity entity = getRegistrationBean().getTableRepo().save(table);
        System.out.println("entity id " + entity);
    }
    @Test
    public void testFindByTableState() {
        List<TableEntity> tableList = getRegistrationBean().getTableRepoImpl().findByTableState(0);
        System.out.println("tableList size ***** " + tableList.size());
    }
}
```

With custom repository, you can implement functionality such as `getrevisionHistory()`. Additionally, spring data support `@Query` annotation and derived query. Here, samples are attached for 2 entities (`DrinkEntity`, `TableEntity`) which are later implemented with spring data.

[[cookbook-spring-data\_query-creation in spring data jpa]] == Query Creation in Spring Data JPA

Below are the ways to create a query in Spring Data JPA:

- Query creation by method names: `List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);` Above method is equivalent to the below query: `select u from User u where u.emailAddress = ?1 and u.lastname = ?2` This is explained in the next section.

- Using JPA Named Queries Example: `@NamedQuery(name = "Drink.nonalcholic", query = "select drink from DrinkEntity drink where drink.alcoholic=false")`
- Using `@Query` annotation

```
@Query(name = "table.query1", value = "select table from TableEntity table where table.state= :#{#criteria.state}")
public Page<TableEntity> findTablesDummy(@Param("criteria") TableSearchCriteriaTo criteria, Pageable pageable);
```

Include above method in repository i.e TableRepo.

- Native Queries - This Queries can be created using `@Query` annotation and setting `nativeQuery=true`
- Similar to the criteria, you have Predicate from QueryDsl. This is explained in below section.

[[cookbook-spring-data\_query-creation by method names]] == Query creation by method names

Consider tablemanagement as an example. First, you will create a TableEntity class with attribute number, waiterId and state. To test query creation by method names, you will create new method `findByState(TableState state)` in TableRepo. This method will find table based on TableState provided. Follow below steps:

- Create TableEntity class as below:

```
@Entity
// Table is a reserved word in SQL/RDBMS and can not be used as table name
@javax.persistence.Table(name = "RestaurantTable")
public class TableEntity extends ApplicationPersistenceEntity implements Table {
    private static final long serialVersionUID = 1L;
    private Long number;
    private Long waiterId;
    private TableState state;
    @Override
    @Column(unique = true)
    public Long getNumber() {
        return this.number;
    }
    @Override
    public void setNumber(Long number) {
        this.number = number;
    }
    @Override
    @Column(name = "waiter_id")
    public Long getWaiterId() {
        return this.waiterId;
    }
    @Override
    public void setWaiterId(Long waiterId) {
        this.waiterId = waiterId;
    }
    @Override
    public TableState getState() {
        return this.state;
    }
    @Override
    public void setState(TableState state) {
        this.state = state;
    }
}
```

- In TableRepo create `findByState(TableState state)` method as below:

```
@Repository
```

```
public interface TableRepo extends JpaRepository<TableEntity, Long>, TableRepoCustom {  
    // Query Creation By method names  
    List<TableEntity> findByState(TableState state);  
}
```

- You will have RegistrationBean class as shown in the previous example. Now, you are ready to test the method `findByState(TableState state)`. In test class, include below test method:

```
@Test  
public void testFindTableByState() {  
    List<TableEntity> tableList = getRegistrationBean().getTableRepo().findByState(TableState.FREE);  
    System.out.println("tableList size " + tableList.size());  
}
```

[[cookbook-spring-data\_implementing-query with querydsl]] == Implementing Query with QueryDsl

Like the JPA Criteria API, it uses a Java 6 annotation processor to generate meta-model objects and produces a much more approachable API. Another good thing about the project is that, it not only has the support for JPA but also allows querying Hibernate, JDO, Lucene, JDBC and even plain collections.

- To start with QueryDsl add below plugin in a pom.xml:

```
<plugin>  
    <groupId>com.mysema.maven</groupId>  
    <artifactId>apt-maven-plugin</artifactId>  
    <version>1.1.1</version>  
    <executions>  
        <execution>  
            <phase>generate-sources</phase>  
            <goals>  
                <goal>process</goal>  
            </goals>  
            <configuration>  
                <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>  
            </configuration>  
        </execution>  
    </executions>  
</plugin>
```

- Execute `mvn clean install` on the project. This will create special query classes e.g for DrinkEntity class generated will be QDrinkEntity.
- To execute Querydsl predicates, you simply let your repository extend `QueryDslPredicateExecutor<T>` Example:

```
@Repository  
public interface DrinkRepo  
    extends JpaRepository<DrinkEntity, Long>, QueryDslPredicateExecutor<DrinkEntity>, DrinkRepoCustom {  
  
    /**  
     * {@inheritDoc}  
     */  
    @Override  
    <S extends DrinkEntity> S save(S entity);  
  
}
```

- You will have registrationBean class, which have above repository autowired in it.
- Create test class and below method.

```
@Test
```

```
public void testFindNonAlcoholicDrinks() {  
  
    QDrinkEntity drinkEntityEqu = QDrinkEntity.drinkEntity;  
    BooleanExpression drink = drinkEntityEqu.alcoholic.isTrue();  
    List<DrinkEntity> drinkList = (List<DrinkEntity>)  
        getDrinkEntityRegistrationBean().getDrinkRepo().findAll(drink);  
    for (DrinkEntity drink1 : drinkList) {  
        System.out.println("drink id " + drink1.getId() + " description: " + drink1.getDescription());  
    }  
}
```

This will return list of drink entities which are nonalcoholic.

[[cookbook-spring-data\_paging-and sorting support]] == Paging and Sorting Support

- For Paging and Sorting support in Spring Data JPA, you should implement PagingAndSortingRepository. Create an interface as shown below:

```
@Repository  
public interface TableRepo extends JpaRepository<TableEntity, Long>, TableRepoCustom {  
    /**  
     * {@inheritDoc}  
     */  
    @Override  
    <S extends TableEntity> S save(S table);  
  
    TableEntity findByNumber(long number);  
    /**  
     * {@inheritDoc}  
     */  
    @Override  
    Page<TableEntity> findAll(Pageable pageable);  
    @Query(name = "table.query", value = "select table from TableEntity table where table.state= ?1")  
    Page<TableEntity> findByTableState1(TableState state, Pageable pageable);  
}
```

- Create test method as below:

```
@Test  
public void testFindTableByState1()  
{  
    PageRequest pageRequest = new PageRequest(0, 2, Direction.DESC, "state");  
    Page<TableEntity> pageEntity =  
        getRegistrationBean().getTableRepo().findByTableState1(TableState.FREE, pageRequest);  
    List<TableEntity> tableList = pageEntity.getContent();  
    for (TableEntity table : tableList) {  
        System.out.println("Table details: " + table.getId() + " , " + table.getWaiterId() + " , " +  
            table.getState());  
    }  
}
```

In the above example, you are extending JpaRepository which in turn extends PagingAndSortingRepository. So, you will get paging and sorting functionality. For Paging and Sorting support, you need to pass Pageable as method Parameter.

```
PageRequest pageRequest = new PageRequest(0, 2, Direction.DESC, "state");  
  
//Here 0 - indicate page number.  
//2 - object on a page  
//Direction Desc or ASC- Sorting sequence Desc or Asc  
//State - this is a property based on which query gets sorted
```

For creating pageRequest object, you have different constructors available as below:

```
PageRequest(int page,int size)
PageRequest(int page,int size,int sort)
PageRequest(int page,int size,Direction direction)
PageRequest(int page, int size, Direction direction, String... properties)
```

### 3.12.2 References

<https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/>  
<http://docs.spring.io/spring-data/jpa/docs/1.4.1.RELEASE/reference/html/jpa.repositories.html>  
<http://javabeat.net/spring-data-jpa-querydsl-integration/>

<http://>  
<http://>

## 3.13 WebSocket

WebSocket is a protocol providing full-duplex communication channels over a single TCP connection. WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application.

The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request. The WebSocket protocol makes more interaction between a browser and a website possible, facilitating the real-time data transfer from and to the server. For more information you can visit [wikipedia](#).

### 3.13.1 WebSocket configuration

In Devonfw a websocket endpoint is configured within the business package as a Spring configuration class. We can find an example in the *salesmanagement* use case of the OASP4J example application.

```
package io.oasp.gastronomy.restaurant.salesmanagement.websocket.config;

...
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/sample");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/websocket/positions").withSockJS();
    }

    ...
}
```

The `@Configuration` annotation indicates that this is a Spring configuration class.

The `@EnableWebSocketMessageBroker` annotation makes Spring Boot registering this endpoint and enables the WebSocket support.

The `AbstractWebSocketMessageBrokerConfigurer` implements the `WebSocketMessageBrokerConfigurer` interface that allows to customize the imported configuration.

The `configureMessageBroker()` method overrides the default method in `WebSocketMessageBrokerConfigurer` to configure the message broker. It starts by calling `enableSimpleBroker()` to enable a simple memory-based message broker to carry the messages back to the client on destinations prefixed with `"/topic"`. The `setApplicationDestinationPrefixes` defines the application name using which browser and server will communicate over WebSocket.

The `registerStompEndpoints` registers the `"/websocket/positions"` endpoint and enables SockJS fallback options so that alternative messaging options may be used if WebSocket is not available. SockJS is a Javascript library which provides websocket like object for browsers and provides cross browser compatibility and support for STOMP protocol. SockJS works in the way that we need to

---

provide URL to connect with message broker and then get the stomp client to communicate. [STOMP](#) is Streaming Text Oriented Messaging Protocol. A STOMP client communicates to a message broker which supports STOMP protocol. STOMP uses different commands like connect, send, subscribe, disconnect, etc. to communicate.

You can see a complete example of how to build an interactive web application using WebSocket [here](#)

## 3.14 File Upload and Download

Apache CXF is an open source services framework. CXF helps you to build and develop services using front-end programming APIs, like JAX-WS and JAX-RS.

[[cookbook-file-upload-and-download\_file-download from cxf]] == File download from CXF

**org.apache.cxf** provides the option to download files of different MIME (Multipurpose Internet Mail Extensions) types.

### Example:

In JAX-RS, annotate the service method with `@Produces("application/octet-stream")`. You can define an interface for the service and annotate it with JAX-RS annotations:

```
public interface DownloadService {  
    @SuppressWarnings("javadoc")  
    @Produces("application/octet-stream")  
    @GET  
    @Path("/downloadFile")  
    public Response getDownloadableFile() throws SQLException, IOException;  
}
```

And here is a simple implementation of the service:

```
@Override  
public Response getDownloadableFile() throws SQLException, IOException {  
    // FILE_PATH - specifies the location of the file in the file system.  
    File file = new File("FILE_PATH");  
    ResponseBuilder response = Response.ok((Object) file);  
    // FILENAME.FILE_EXTENSION - specifies the filename and its extension after download.  
    response.header("Content-Disposition", "attachment; filename=FILENAME.FILE_EXTENSION");  
    return response.build();  
}
```

In the above code snippet, a file object is constructed by supplying path of the file to be downloaded. ResponseBuilder is created from this file object and finally the response is built from ResponseBuilder object.

Following table explains the annotations used for the download functionality:

Table 3.5. Annotations used for download functionality

Annotation	Description
<code>@Produces</code>	It is used to specify the MIME media type that can be produced and sent back to the client.
<code>@GET</code>	It is a request method designator defined by JAX-RS and corresponds to the similarly named HTTP method.
<code>@PATH</code>	This identifies the URI path template to which the resource responds, and is specified at the class level of a resource.

### 3.14.1 Produces Annotation

`@Produces` annotation is used to specify the MIME media type that can be produced and sent back to the client. If `@Produces` is applied at the class level, all the methods in a resource produce the

specified MIME type by default. If applied at the method level, the annotation overrides any @Produces annotation applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the JAX-RS runtime sends back an HTTP "406 Not Acceptable" error.

The value of @Produces is an array of String of MIME types. For example: @Produces({"image/jpeg,image/png"})

The following example shows how to apply @Produces at both, the class and method levels:

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
    ...
    }
    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
    ...
    }
}
```

[[cookbook-file-upload-and-download\_file-upload from cxf]] == File upload from CXF

**org.apache.cxf** provides option to upload files of different MIME (Multipurpose Internet Mail Extensions) type.

#### Example:

In JAX-RS, annotate the service method with @Consumes("multipart/mixed"). You can define an interface for the service and annotate it with JAX-RS annotations:

```
import javax.ws.rs.core.MediaType;

public interface UploadService {
    @SuppressWarnings("javadoc")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    @GET
    @Path("/uploadFile")
    public Response uploadFile(List<Attachment> attachments) throws SQLException, IOException;
}
```

And here is a simple implementation of the service:

Add annotation @Transactional at class level and inject UcManageBinaryObject class object.

```
@Transactional
public class implClassName {

    @Inject
    private UcManageBinaryObject ucManageBinaryObject;

    /**
     * @return ucManageBinaryObject
     */
    public UcManageBinaryObject getUcManageBinaryObject() {

        return this.ucManageBinaryObject;
    }
}
```

## Web Service method implementation:

```
@POST
@Consumes ("multipart/mixed")
// use @Consumes(MediaType.MULTIPART_FORM_DATA) if request contains multi-part form data and import // 
jaxws.ws.rs.core.MediaType;
@Path ("~/upload")
public Response uploadFile(List<Attachment> attachments) {
    BinaryObjectEto binaryObject = new BinaryObjectEto();
    Blob blob = null;
    for (Attachment attachment: attachments) {
        DataHandler handler = attachment.getDataHandler();
        try {
            InputStream stream = handler.getInputStream();
            OutputStream outputStream = new ByteArrayOutputStream();
            IOUtils.copy(stream, outputStream);
            byte[] byteArray = outputStream.toString().getBytes();
            if (byteArray != null && byteArray.length != 0) {
                blob = new SerialBlob(byteArray);
                getUcManageBinaryObject().saveBinaryObject(blob, binaryObject);
            }
        } catch (SQLException e) {
            throw new SQLException(e.getMessage(), e);
        } catch (IOException e) {
            throw new IOException(e.getMessage(), e);
        }
    }
    return Response.ok("file uploaded").build();
}
```

In the above code snippet, uploaded attachments are iterated and InputStream for each attachment is extracted. Each InputStream is converted into the byte array and a Blob object is created out of it. The Blob object is saved into the database by calling saveBinaryObject(blob, binaryObject).

Following table explains the annotations used for upload functionality:

Table 3.6. Annotations used for upload functionality

Annotation	Description
@Consumes	It is used to specify MIME media types that can be accepted, or consumed, from the client.
@GET	It is a request method designator defined by JAX-RS and corresponds to the similarly named HTTP method.
@PATH	This identifies the URI path template to which the resource responds, and is specified at the class level of a resource.

### 3.14.2 Consumes Annotation

The @Consumes annotation is used to specify MIME media types that can be accepted, or consumed, from the client. If @Consumes is applied at the class level, all the response methods accept the specified MIME type by default. If applied at the method level, @Consumes overrides any @Consumes annotation applied at the class level.

If a resource is unable to consume the MIME type of a client request, the JAX-RS runtime sends back an HTTP 415 ("Unsupported Media Type") error.

The value of `@Consumes` is an array of String of acceptable MIME types. For example:  
`@Consumes({"text/plain,text/html"})`

The following example shows how to apply `@Consumes` at both, the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

### 3.14.3 MIME Types

MIME stands for "Multipurpose Internet Mail Extensions". It is a way of identifying files on the Internet, according to their nature and format. For example, using the "Content-type" header value defined in an HTTP response, the browser can open the file with the proper extension/plugin.

For more information, visit: <http://www.freeformatter.com/mime-types-list.html>

## 3.15 Microservices in Devonfw

### 3.15.1 Introduction

The Microservices architecture is an approach for application development based on a series of *small* services grouped under a business domain. Each individual service runs autonomously and communicating with each other through their APIs. That independence between the different services allows to manage (upgrade, fix, deploy, etc.) each one without affecting the rest of the system's services. In addition to that the microservices architecture allows to scale specific services when facing an increment of the requests, so the applications based on microservices are more flexible and stable, and can be adapted quickly to demand changes.

However, this new approach, developing apps based on microservices, presents some downsides. To address those, Devonfw microservices approach is based on [Spring Cloud Netflix](#), that provides all the main components for microservices integrated within Spring Boot context.

Let's see the main challenges when working with microservices:

- Having the applications divided in different services we will need a component (router) to redirect each request to the related microservice. These redirection rules must implement filters to guarantee a proper functionality.
- In order to manage correctly the routing process, the application will also need a catalog with all the microservices and its details: IPs and ports of each of the deployed instances of each microservice, the state of each instance and some other related information. This catalog is called *Service Discovery*.
- With all the information of the *Service Discovery* the application will need to calculate and select between all the available instances of a microservice which is the suitable one. This will be figured out by the library *Client Side Load Balancer*.
- The different microservices will be likely interconnected with each other, that means that in case of failure of one of the microservices involved in a process, the application must implement a mechanism to avoid the error propagation through the rest of the services and provide an alternative as a process result. To solve this, the pattern *Circuit Breaker* can be implemented in the calls between microservices.
- As we have mentioned, the microservices will exchange calls and information with each other so our applications will need to provide a secured context to avoid not allowed operations or intrusions. In addition, since microservices must be able to operate in an isolated way, it is not recommended to maintain a session. To meet this need without using Spring sessions, a token-based authentication is used that exchanges information using the [json web token \(JWT\)](#) protocol.

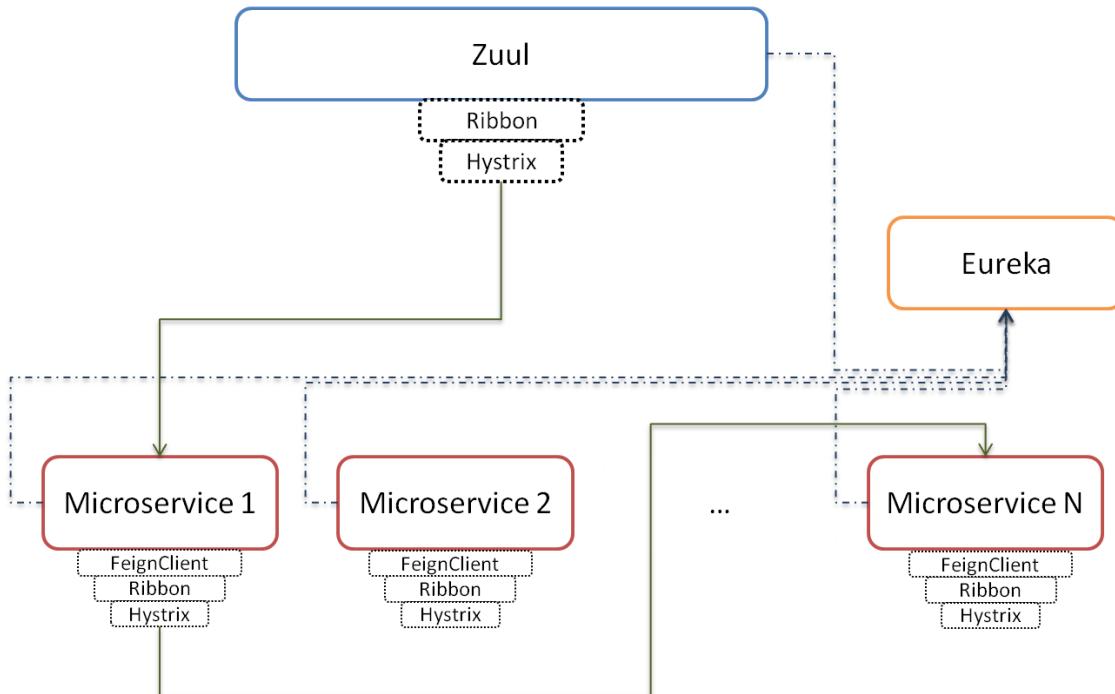
In addition to all of this we will find other issues related to this particular architecture that we will address fitting the requirements of each project.

- Distributed data bases: each instance of a microservice should have only one data base.
- Centralized logs: each instance of a microservice creates a log and a trace that should be centralized to allow an easier way to read all that information.
- Centralized configuration: each microservice has its own configuration, so our applications should group all those configurations in only one place to ease the configuration management.

- Automatized deployments: as we are managing several components (microservices, catalogs, balancers, etc.) the deployment should be automatized to avoid errors and ease this process.

### 3.15.2 Microservices schema

In the following schema we can see an overview of the structure of components in a Devon application based on the *Spring Cloud Netflix* solution for microservices.



Let's explain each component

[[devon-microservices\_service-discovery - eureka]] === Service Discovery - Eureka

[Eureka](#) is a server to register and locate the microservices. The main function for *Eureka* is to register the different instances of the microservices, its location, its state and other metadata.

It works in a simple way, during the start of each microservice, this communicates with the *Eureka* server to notify its availability and to send the metadata. The microservice will continue to notify its status to the Eureka server every 30 seconds (default time on Eureka server properties). This value can be changed in the configuration of the component.

If after 3 periods, Eureka does not receive notification of any of the microservices, it will be considered as unavailable and will eliminate its registration.

In addition, it serves as a catalog to locate a specific microservice when routing a request to it.

[[devon-microservices\_circuit-breaker - hystrix]] === Circuit Breaker - Hystrix

[Hystrix](#) is a library that implements the [Circuit Breaker](#) pattern. Its main functionality is to improve the reliability of the system, isolating the entry points of the microservices, preventing the cascading failure from the lower levels of the application all the way up to the user.

In addition to that, it allows developers to provide a fallback in case of error. *Hystrix* manages the requests to a service, and in case that the microservice doesn't response, allows to implement an alternative to the request.

[[devon-microservices\_client-side load balancer - ribbon]] === Client Side Load Balancer - Ribbon

[Ribbon](#) is a library designed as client side load balancer. Its main feature is to integrate with *Eureka* to discover the instances of the microservices and their metadata. In that way the *Ribbon* is able to calculate which of the available instances of a microservice is the most appropriate for the client, when facing a request.

[[devon-microservices\_rest-client - feign]] === REST Client - Feign [Feign](#) is a REST client to make calls to other microservices. The strength of Feign is that it integrates seamlessly with *Ribbon* and *Hystrix*, and its implementation is through annotations, which greatly facilitates this task to the developer.

Using annotations, Spring-cloud generates, automatically, a fully configured REST client.

[[devon-microservices\_router-and filter - zuul]] === Router and Filter - Zuul

[Zuul](#) is the entry point of the apps based on Spring-cloud microservices. It allows dynamic routing, load balancing, monitoring and securing of requests. By default *Zuul* uses *Ribbon* to locate, through *Eureka*, the instances of the microservice that it wants to invoke and sends the requests within a *Hystrix Command*, taking advantage of its functionality.

[[devon-microservices\_how-to create microservices in devonfw]] == How to create microservices in Devonfw

In order to generate microservices in a Devonfw project we can choose between two approaches:

- generate a new OASP4J application and implement one by one all the needed components (based on Spring Cloud).
- generate a new OASP4J application through the custom microservice archetype included in the Devonfw distributions.

That second approach, using the Devonfw microservices archetype, will generate automatically all the basic structure and components to start developing the microservices based application.

## Devonfw archetypes

To simplify starting with projects based on microservices, Devonfw includes two archetypes to generate pre-configured projects that include all the basic components of the *Spring Cloud* implementation.

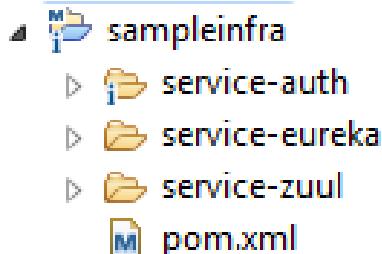
- **archetypes-microservices-infra**: generates a project with the needed infrastructure services to manage microservices. Includes the *Eureka* service, *Zuul* service and the authentication service.
- **archetypes-microservices**: generates a simple project pre-configured to work as a microservice.

[[devon-microservices\_create-new microservices infrastructure application]] === Create New Microservices infrastructure application

To generate a new microservices infrastructure application through the Devonfw archetype you only need to open a Devonfw console (*console.bat* script) and follow the same steps described in [getting started creating new devonfw application](#). But, instead of using the *standard* archetype, we must provide the special infrastructure archetype *archetype-microservice-infra*. Remember to provide your own values for *DgroupId*, *DartifactId*, *Dversion* and *Dpackage* parameters:

```
mvn -DarchetypeVersion=2.2.0 -DarchetypeGroupId=com.capgemini.devonfw.microservices -  
DarchetypeArtifactId=microservices-infra-archetype archetype:generate -DgroupId=com.capgemini -  
DartifactId=sampleinfra -Dversion=0.1-SNAPSHOT -Dpackage=com.capgemini.sampleinfra
```

Once the *Maven* command has finished an application with the following modules should be created:



#### service-eureka module

This module contains the needed classes and configuration to start a *Eureka* server.

This service runs by default on port 8761 although it can be changed in the *application.properties* file of the project.

#### service-zuul module

This module contains all the needed classes and configuration to start a *Zuul* server, that will be in charge of the routing and filter of the requests.

This service by default runs on port 8081 but, as we already mentioned, it can be changed through the *file application.properties* of the project.

#### service-auth module

This module runs an authentication and authorization microservice that allows to generate a security token to make calls to the rest of microservices. This module is only providing a basic structure, the security measures must be implemented fitting the requirements of each project.

This service runs by default on port 9999, although, as in previous services, it can be edited in the *application.properties* file.

[[devon-microservices\_create-new microservices application]] === Create New Microservices Application

To generate a new microservice project through the Devonfw archetype, as in previous archetype example, you can follow the instructions explained in [getting started creating new devonfw application](#). But, instead of using the *standard* archetype, we must provide the special microservices archetype *archetype-microservices*. Open a Devonfw console (*console.bat* script) and launch a *Maven* command like the following (provide your own values for *DgroupId*, *DartifactId*, *Dversion* and *Dpackage* parameters):

```
mvn -DarchetypeVersion=2.2.0 -DarchetypeGroupId=com.capgemini.devonfw.microservices -  
DarchetypeArtifactId=microservices-archetype archetype:generate -DgroupId=com.capgemini -  
DartifactId=sampleappl -Dversion=0.1-SNAPSHOT -Dpackage=com.capgemini.sampleappl
```

That command generates a simple application containing the source code for the microservice. By default, the *pom.xml* includes the *devon-microservices* module, that contains the security configuration, jwt interceptors, *Hystrix*, *Ribbon* and *FeignClient* configuration and some properties common to all microservices.

The created microservice runs by default on port `9001` and has the `context-path` with the same name than the project. This parameters can be changed through the `'application.properties'` file of the project.

[[devon-microservices\_how-to use microservices in devonfw]] == How to use microservices in Devonfw

In the following sections we are going to provide some patterns to manage microservices in Devonfw using the archetype, alongside the options that each of the available modules offer.

### Eureka service

We are going to review the general options for the *Eureka* service. If you are interested in getting more details you can visit the official site for [Spring Cloud Eureka clients](#).

To create an *Eureka* server you only need to create a new *Spring Boot* application and add the `@EnableEurekaServer` to the main class.

#### Note

The provided archetype `archetype-microservices-infra` already provides that annotated class.

```
@Configuration
@EnableEurekaServer
@EnableAutoConfiguration
@SpringBootApplication
public class EurekaBootApp {

    public static void main(String[] args) {
        new SpringApplicationBuilder(EurekaBootApp.class).web(true).run(args);
    }
}
```

The basic properties that must be configured for *Eureka* server are:

- port: in which port the service will run. The default port is the `8761` and you have to keep in mind that the connection to this port is specially critical as all the microservices must be able to connect to this `host:port`. Remember that *Eureka* generates and manages the microservices catalog, so it's crucial to allow the microservices to register in this component.
- url: which *URL* manages as area.

```
eureka.instance.hostname=localhost
eureka.instance.port=8761

server.port=${eureka.instance.port}

eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${eureka.instance.port}/eureka/
```

The way to connect a microservice to *Eureka* server is really simple. You only will need to specify the `host:port` where the server is located and annotate the *Spring Boot* class with `@EnableMicroservices` annotation.

#### Note

Instead of using that `@EnableMicroservices` annotation, you can use the equivalent *Spring* annotations `@EnableDiscoveryClient` or `@EnableEurekaClient`.

```
@Configuration
@EnableMicroservices
@SpringBootApplication
public class MicroserviceBootApp {
    public static void main(String[] args) {

        SpringApplication.run(MicroserviceBootApp.class, args);
    }
}
```

```
eureka.instance.hostname=localhost
eureka.instance.port=8761

eureka.client.serviceUrl.defaultZone=http://{$eureka.instance.hostname}:{$eureka.instance.port}/eureka/
```

With this the application will register automatically in *Eureka* and will be validated each 30 seconds. This value can be changed editing the property `eureka.instance.leaseRenewalIntervalInSeconds` in `application.properties` file. It must be taken into account that each *Eureka* client will maintain a cache of *Eureka* records to avoid calling the service every time it is necessary to access another microservice. This cache is reloaded every 30 seconds, this value can also be edited through property `eureka.client.registryFetchIntervalSeconds` in `application.properties` file.

## Zuul service

We are going to show an overview to the options of the *Zuul* service, if you want to know more details about this particular service visit the official site of [Spring Cloud](#).

*Zuul* is the component in charge for router and filtering the requests to the microservices system. It works as a gateway that, through a rule engine, redirects the requests to the suitable microservice. In addition, it can be used as a security filter as it can implement PRE-Filters and POST-Filters.

To create a basic *Zuul* server you only need to create a new Spring Boot application and add the `@EnableZuulProxy` annotation.

```
@EnableAutoConfiguration
@EnableEurekaClient
@EnableZuulProxy
@SpringBootApplication
public class ZuulBootApp {
    public static void main(String[] args) {

        SpringApplication.run(ZuulBootApp.class, args);
    }
}
```

To allow *Zuul* to redirect the requests we need to connect *Zuul* with the previously created *Eureka* service, to allow him to register and access to the catalog of microservices created by *Eureka*.

Also, if we are going to use the *Zuul* service from a web browser, we must configure the CORS filter to allow connections from any source. This is really easy to implement by adding the following Java Bean to our `ZuulBootApp` class:

```
@Bean
public CorsFilter corsFilter() {
    final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    final CorsConfiguration config = new CorsConfiguration();
    config.setAllowCredentials(true);
    config.addAllowedOrigin("*");
    config.addAllowedHeader("*");
    config.addAllowedMethod("OPTIONS");
```

```
        config.addAllowedMethod("HEAD");
        config.addAllowedMethod("GET");
        config.addAllowedMethod("PUT");
        config.addAllowedMethod("POST");
        config.addAllowedMethod("DELETE");
        config.addAllowedMethod("PATCH");
        source.registerCorsConfiguration("/**", config);
        return new CorsFilter(source);
    }
```

To configure the *Zuul* service we need to define a series of properties that we will describe below:

```
server.port=8081
spring.application.name=zuulserver

eureka.instance.hostname=localhost
eureka.instance.port=8761
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${eureka.instance.port}/eureka/

microservices.context-path=/demo

zuul.routes.security.path=${microservices.context-path}/services/rest/security/**
zuul.routes.security.serviceId=AUTH
zuul.routes.security.stripPrefix=false

zuul.routes.login.path=${microservices.context-path}/services/rest/login
zuul.routes.login.serviceId=AUTH
zuul.routes.login.stripPrefix=false

zuul.ignoredServices='*'
zuul.sensitive-headers='

ribbon.eureka.enabled=true
hystrix.command.default.execution.timeout.enabled=false
```

- `server.port`: Is the port where the *Zuul* service is listening.
- `spring.application.name`: The name of the service the will be sent to *Eureka*.
- `eureka.*`: The properties for the register of the *Eureka* client.
- `zuul.routes.XXXXX`: The configuration of a concrete route.
- `zuul.routes.XXXXX.path`: The path used for a redirection.
- `zuul.routes.XXXXX.serviceId`: ID of the service where the request will be redirected. It must match the property `spring.application.name` in the microservice.
- `zuul.routes.XXXXX.stripPrefix`: by default set to `false`. With this property we configure if the part of the route that has matched the request must be *cutted out*. i.e., if the path is `/sample/services/rest/foomanagement/##` and the property is set to `true` it will redirect to the microservice but it will only send the path `**`, the root `/sample/services/rest/foomanagement/` will be removed.
- `zuul.ignoredServices`: Configures which services without result in the routes, must be ignored.
- `zuul.sensitive-headers`: Configures which headers must be ignored. This property must be set to `empty`, otherwise *Zuul* will ignore security authorization headers and the json web token will not work.
- `ribbon.eureka.enabled`: Configures if the *Ribbon* should be used to route the requests.

- `hystrix.command.default.execution.timeout.enabled`: Enables or disables the timeout parameter to consider a microservices as unavailable. By default the value for this property is 1 second. Any request that takes more than this will be considered failed. By default in the archetype this property is disabled.

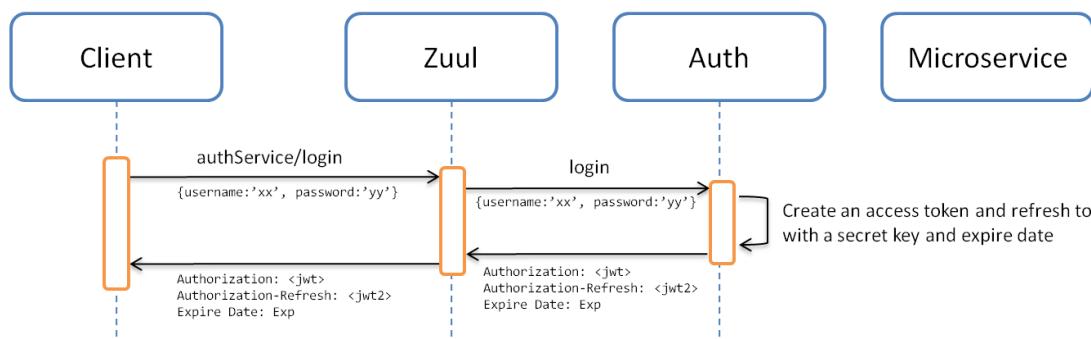
Having an *Eureka* client activated, the *Zuul* service will refresh its content every 30 seconds, so a just registered service may still have not been cached in *Zuul*. On the contrary, if a service is unavailable, 3 cycles of 30 seconds must pass before *Eureka* sets its register as *dead*, and other 30 seconds for *Zuul* to refresh its cache.

### Security, Authentication and authorization

The most commonly used authentication in micro-service environments is authentication based on [json web tokens](#), since the server does not need to store any type of user information (stateless) and therefore favors the scalability of the microservices.

It works as follows:

- The user is authenticated in our application, either through a user / password access, or through a third provider.
- This authentication request is launched against the *Zuul* server which will redirect it to an instance of the *Auth* microservice.
- The *Auth* microservice will check the user, retrieve their roles and metadata and generate two tokens: one with user access information and another needed to refresh the access token. This information will be returned to the customer.



the `service-auth` service is already prepared to listen to the `/login` path and generate the two mentioned tokens. To do so we can use the `JsonWebTokenUtility` class that is implemented in Devonfw

```

UserDetailsJsonWebTokenAbstract clientTo = new UserDetailsJsonWebTokenTo();
clientTo.setId(1L);
clientTo.setUsername("demo");
clientTo.setRoles(new ArrayList<>(Arrays.asList("DEMO")));
clientTo.setExpirationDate(buildExpirationDate(this.expirationTime * 60 * 1000L));

return new ResponseEntity<>(new
JwtHeaderTo(this.jsonWebTokenUtility.createJsonWebTokenAccess(clientTo),
this.jsonWebTokenUtility.createJsonWebTokenRefresh(clientTo),
this.expirationTime * 60 * 1000L), HttpStatus.OK);
  
```

### Note

In our example you can make a POST request to:

<http://localhost:8081/service-auth/services/rest/login>

HEADER Content-Type : application/json  
 BODY { "j\_username" : "xxx", "j\_password" : "xxx"}

This will generate a response like the following

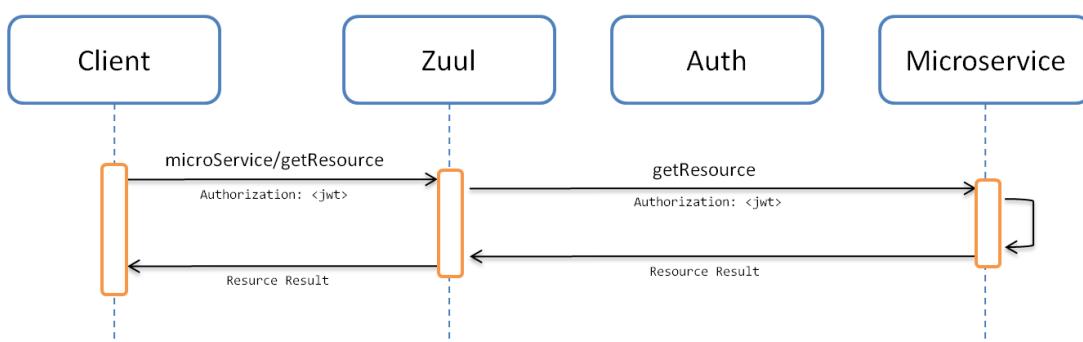
```
{
  "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkZW1vIiwizmlyc3ROYW1IijoizGVtbyIsImxhc3ROYW1IijoizGVtbyIsImV4CCI6MTQ4Nzg3Uw1-ZGf2WF58qjWc4xg",
  "refreshToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkZW1vIiwizmlyc3ROYW1IijoizGVtbyIsImxhc3ROYW1IijoizGVtbyIsImV4CCI6MTQ4Nzg3h1G7syTK36YHxkGniyiTlxnazZXi8tT-RtUxxW8We8cdiYJn6tw0RoFkOyr1F5EzvkGyU0HNoLyw",
  "expirationTime": 900000,
  "accessHeaderName": "Authorization",
  "refreshHeaderName": "Authorization-Refresh"
}
```

The client now should store, in the header defined in `accessHeaderName`, the token included as `accessToken`. By default, when using [devon4sencha](#), this functionality is already implemented.

When configuring the `service-auth` module is very important to have into account the following aspects:

- The *expiration date* of the token can be configured in the properties file with the property `jwt.expirationTime` (will appear in minutes).
- The key for the token generation can be configured also in the properties file using the property `jwt.encodedKey` which will have a *Base64* encoded value.
- The roles inserted in the token should be the list of the access roles of the user. Doing this we avoid that each microservice has to look for the roles that belong to a profile.
- If you want to use a specific `UserDetails` for the project, with new fields, you must extend the behavior as explained in [here](#).

From now on, the client will be able to make calls to the microservices, sending the *access token* in the header of the request.



Once the request reaches the microservice, the app must validate the token and register the user in the security context. These operations will be automatic as long as the microservice has enabled the security inherited from the `JsonWebTokenSecurityConfig` class. This is done using the following code:

```

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends JsonWebTokenSecurityConfig {

```

```

@Override
public JsonWebTokenUtility getJsonWebTokenUtility() {
    return new JsonWebTokenUtility();
}

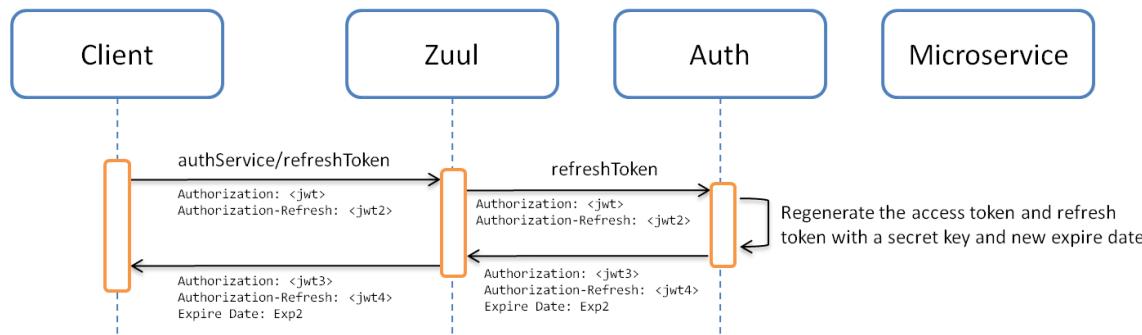
@Override
protected void setupAuthorization(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        // authenticate all other requests
        .anyRequest().authenticated();
}

}

```

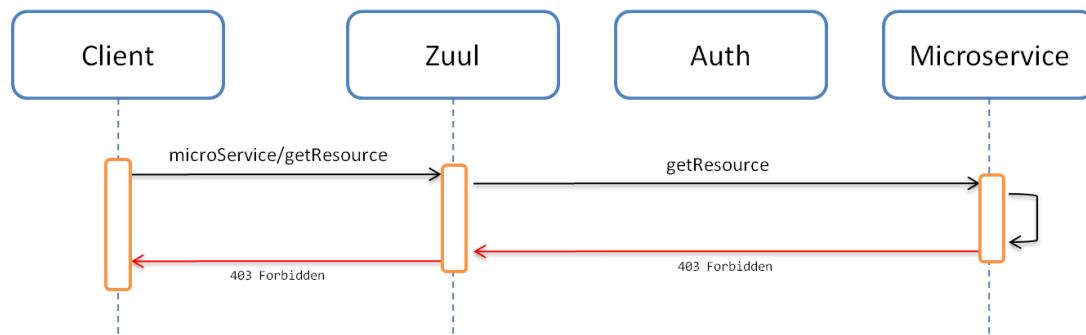
In addition, Devonfw has already implemented the needed interceptors and filters to resend the security header each time that a microservice calls other microservice of the ecosystem.

When validating the token, it is also checked its expiration date, so it is highly recommended that the client refresh from time to time the token, in order to update its expiration date. This is done by launching a request to `/refresh_jwt` within the `service-auth` module and sending both the *access token* and the *refresh token* in the header.



When using [devon4sencha](#) that requests are automated by the framework.

If for any reason an attempt is made to access a business operation without having a valid token, or without sufficient *role* level permission to execute that operation, the microservice response will be **Forbidden**.



[[devon-microservices\_how-to modify the userdetails information]] ===== How to modify the `UserDetails` information

In order to modify the `UserDetails` information we will need to accomplish two steps: modify the authentication service to generate the authentication token with the custom attributes embedded, and

modify the pre-authentication filter of the microservices to convert the token into an *Object* with the custom attributes available.

[[devon-microservices\_modify-the authentication service to generate a new token]] ===== Modify the authentication service to generate a new token

We must modify the `service-auth` that is in charge of logging the user and generate the security token.

The first thing to do is to create a `UserDetails` class that contains the required attributes and custom attributes. In the code sample we will call this class `UserDetailsJsonWebTokenCustomTo`, and must either implement the generic `UserDetailsJsonWebTokenAbstract` interface or extend it from the current `UserDetailsJsonWebTokenTo` class, since the services are prepared to work with it. In the example, we will add two new attributes `firstName` and `lastName`.

```
public class UserDetailsJsonWebTokenCustomTo extends UserDetailsJsonWebTokenTo {

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

In case that the `UserDetailsJsonWebTokenAbstract` interface is implemented, in addition to the new attributes the rest of the interface must be implemented.

The next step would be to override the component that performs the conversions `Token#UserDetails` and `UserDetails#Token`. This component is the `JsonWebTokenUtility`, so you should create a new class that extends from this, in the example we will call it `JsonWebTokenUtilityCustom`. In this new class, you must overwrite the only two methods that are allowed to perform the conversions, to add *writing* and *reading* operations for the new custom attributes.

```
public class JsonWebTokenUtilityCustom extends JsonWebTokenUtility {

    @Override
    protected UserDetailsJsonWebTokenAbstract addCustomPropertiesClaimsToUserDetails(Claims claims) {
        UserDetailsJsonWebTokenCustomTo userDetails = new UserDetailsJsonWebTokenCustomTo();

        userDetails.setFirstName(claims.get("firstName", String.class));
        userDetails.setLastName(claims.get("lastName", String.class));

        return userDetails;
    }

    @Override
    protected void addCustomPropertiesUserDetailsToJwt(UserDetailsJsonWebTokenAbstract
        authTokenDetailsDTO, JwtBuilder jBuilder) {

```

```
UserDetailsJsonWebTokenCustomTo userDetails = (UserDetailsJsonWebTokenCustomTo) authTokenDetailsDTO;

jBuilder.claim("firstName", userDetails.getFirstName());
jBuilder.claim("lastName", userDetails.getLastName());
}

}
```

Now you should enable that new converter to replace the default one. In the WebSecurityConfig class you must change the related @Bean to start using this new class

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    ...

    @Bean
    public JsonWebTokenUtility getJsonWebTokenUtility() {
        return new JsonWebTokenUtilityCustom();
    }

    ...

}
```

Finally, in the login process the new attributes should be filled in when creating the user. In our example in the class SecuritymanagementRestServiceImpl.

```
UserDetailsJsonWebTokenCustomTo clientTo = new UserDetailsJsonWebTokenCustomTo();
clientTo.setId(1L);
clientTo.setUsername("demo");
clientTo.setRoles(new ArrayList<>(Arrays.asList("DEMO")));
clientTo.setExpirationDate(buildExpirationDate(this.expirationTime * 60 * 1000L));

clientTo.setFirstName("firstName");
clientTo.setLastName("lastName");

return new ResponseEntity<>(new
JwtHeaderTo(this.jsonWebTokenUtility.createJsonWebTokenAccess(clientTo),
    this.jsonWebTokenUtility.createJsonWebTokenRefresh(clientTo), //
    this.expirationTime * 60 * 1000L), HttpStatus.OK);
```

[[devon-microservices\_modify-the pre-authentication filter to read the new token]] ===== Modify the pre-authentication filter to read the new token

Once a token with custom attributes has been obtained, the steps to read it and put it in the security context are very simple. The changes shown in this point should be reproduced in those microservices where you want to use the new custom attributes. The steps to follow are those:

- Create a UserDetailsJsonWebTokenCustomTo class that contains the new attributes, as was done in the previous section. The ideal would be to reuse the same class.
- Create a JsonWebTokenUtilityCustom class that extends the implementation of the token generator, just as it was done in the previous section. Again, the ideal would be to reuse the same class.
- Configure the creation of this new @Bean in the WebSecurityConfig class just like in the previous section.

With these three steps you can use the new security object with the custom attributes. One way to use it could be as follows:

```
UserDetailsJsonWebToken principal = (UserDetailsJsonWebToken)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();

UserDetailsJsonWebTokenCustomTo userDetails = (UserDetailsJsonWebTokenCustomTo)
principal.getUserDetailsJsonWebTokenAbstract();

userDetails.getFirstName();
```

## [[devon-microservices\_how-to start with a microservice]] === How to start with a microservice

Once the microservice has been created through its archetype, you need to have a series of points in mind to configure it correctly:

- The microservice must have the `microservices` module as dependency in its `pom.xml` configuration to be able to use the interceptors and the generic configuration.

```
<dependency>
  <groupId>com.capgemini.devonfw.microservices</groupId>
  <artifactId>microservices</artifactId>
</dependency>
```

- It should be annotated in its initial class with `@EnableMicroservices`, this will activate the annotations for *Eureka* client, *CircuitBreaker* and the client *Feign*. All of this is configured in the properties file.
- This is a *bootified* application so in the `pom.xml` file you will have to define which one is the boot class.
- You must consider the boot configuration: *port* and *context-path*. In development, each microservice must have a different port, to avoid colliding with other microservices, while the *context-path* is recommended to be the same, to simplify the *Zuul* configurations and calls between microservices.
- You can use `@RolesAllowed` annotations in the services methods to secure them, as long as the Web security inherited from `JsonWebTokenSecurityConfig` has been enabled, since it is the responsible for putting the *UserDetails* generated from the token into the security context.
- All microservices must share the security key to encrypt and decrypt the token. And, specially, it should be the same as the `service-auth`, which will be responsible for generating the initial token.
- In the *Zuul* module, the routes must be well configured to be able to route certain URLs to the new created microservices. So, if we have added a `sampleapp1` with `server.context-path=/sampleapp1` we will need to map that service in the *Zuul*'s `application.properties` file adding

```
zuul.routes.sampleapp1.path=/sampleapp1/services/rest/**  
zuul.routes.sampleapp1.serviceId=sampleapp1  
zuul.routes.sampleapp1.stripPrefix=false
```

The rest will be treated as if it were a normal Web application, which exposes some services through a REST API.

## [[devon-microservices\_calls-between microservices]] === Calls between microservices

In order to invoke a microservice manually, you would need to implement the following steps:

- Obtain the instances of the microservice you want to invoke.
- Choose which of all instances is the most optimal for the client.

- Retrieve the security token from the source request.
- Create a REST client that invokes the instance by passing the generated security token.
- Intercept the response in case it causes an error, to avoid a cascade propagation.

Thanks to the combination of *Feign*, *Hystrix*, *Ribbon*, *Eureka* and *Devonfw* it is possible to make a call to another microservice in a declarative, very simple and almost automatic way.

You only need to create an interface with the methods that need to be invoked. This interface must be annotated with `@FeignClient` and each of the methods created must have a path and a method in the `@RequestMapping` annotation. An example interface might be as follows:

```
@FeignClient(value = "foo")
public interface FooClient {

    @RequestMapping(method = RequestMethod.GET, value = "${server.context-path}/services/rest/
foomanagement/v1/foo")
    FooMessageTo foo();

}
```

It is important to highlight the following aspects:

- The `@FeignClient` annotation comes along with the name of the microservice to be invoked. The correct and optimal would be to use the name of the microservice, but it is also possible to launch the request to the *Zuul* server. In the latter case it would be the server itself that would perform the load balancing and self-discovery of the most appropriate microservice, but have in mind that, doing this, the proxy server is also unnecessarily overloaded with unnecessary requests.
- The `@RequestMapping` annotation must have the same method and path as expected on target, otherwise the request will be thrown and no response will be found.
- The input and output parameters will be mapped to *json*, so they may not be exactly the same classes in both destination and source. It will depend on how you want to send and retrieve the information.

Once the interface is created and annotated, in order to use the calls, it would be enough to inject the component into the object from which we want to use it and invoke any of its methods. *Spring Cloud* will automatically generate the required bean.

```
...
@Inject
FooClient fooClient;

public FooMessageTo invokeFooClient() {
    return this.fooClient.foo();
}
```

With these two annotations, almost all the functionality is covered automatically: search in *Eureka*, choice of the best instance through *Ribbon*, registration of the token and creation of the REST client. Only would be necessary to control the response in case of failure. The idea is to allow, in case of failure or fall of the invoked microservice, from the origin of the invocation is executed an alternative plan. This is as simple as activating the `fallback` in the `@FeignClient` annotation and assigning a class that will be invoked in case the REST client response fails.

```
@FeignClient(value = "foo", fallback = FooClientHystrixFallback.class)
public interface FooClient {

    @RequestMapping(method = RequestMethod.GET, value = "${server.context-path}/services/rest/
foomanagement/v1/foo")
    FooMessageTo foo();

}
```

Finally, you will need to create a class annotated with `@Component` that implements the interface of the *Feign* client. Within this implementation you can add the desired functionality in case the invocation to the REST client fails.

```
@Component
public class FooClientHystrixFallback implements FooClient {

    @Override
    public FooMessageTo foo() {
        return new FooMessageTo("Fail Message");
    }
}
```

## 3.16 Docker

### 3.16.1 Introduction

Docker is an open source project to wrap, ship and run software using containers.

The docker containers run independently of the hardware or platform you are using, and the application wrapped doesn't require to be built on a specific language or framework. That makes them great building blocks for deploying and scaling network applications like web servers, databases, mail servers with a small amount of effort.

[[cookbook-dockerization\_install-docker on windows]] == Install Docker on Windows

If you do not use Windows 10, you can not run Docker directly on your machine. Rather, you need a virtual machine, with a specific Linux distribution on it. Docker offers a ready to use [toolbox](#) that automates this process. In order to run a virtual machine, your machine needs virtualization enabled. This can be done in your BIOS settings. For a Lenovo T450, this works as shown below. This will work similar for other models:

- Shut down your computer
- Turn on your computer and keep hitting the button F1
- You will hear some kind of beep and the BIOS screen will pop up
- Navigate to the Security tab, then Virtualization and hit enter
- Enable Intel ® Virtualization Technology and hit F10 and confirm the changes by hitting 'Y' option
- Boot your machine

Now, install Docker Toolbox (full installation is recommended). When the installation is done, start Docker Quickstart Terminal to see if everything works. As soon as you see some whale and a command prompt, you can try the following command: `docker run hello-world`. This will download and run a container from docker hub and print some message, if this was successful.

[[cookbook-dockerization\_use-docker manually]] == Use Docker manually

As docker (boot2docker) runs on virtualized machine, you may need to share folders with this virtual machine (VM). Start Oracle VM VirtualBox. You should see a machine called default. If you have started the Docker Quickstart Terminal prior to this, then the `default` should be running. In order to add a shared folder, you have to shut down the machine. You can do this by right clicking on `default`, `close`, `power off`. Then, click on `default` and go to `Settings`, `Shared Folders`. On the right side, you can click on `Adds new shared folder`. In the folder path, go to `Other` and navigate to the folder on your machine that you wish to share with the VM (e.g. `c:\dockershare`) and type a name (e.g. `dockershare`). Toggle `Automount` and `Make Permanent` and save the changes. Now, you can either start the VM out of Oracle VM VirtualBox, or again start the Docker Quickstart Terminal. It is advised to select the second option, as you can keep your keyboard layout in the language of the host machine. If you have chosen to start the Quickstart Terminal, then ssh into the VM with `docker-machine ssh default`. From the Linux VM, you can execute docker commands, as docker is installed on that distribution. Now, create a folder which contains the shared files and mount the shared folder:

```
sudo mkdir /dockershare
sudo mount -t vboxsf dockershare /dockershare
cd /dockershare/
```

If there are files in this folder on your Windows host, you can see them with `ls`. To manually create a container with the sample application, copy a bootable `oasp4j-sample-server.war` in the shared folder. Docker uses `Dockerfile` to create images, which can be started as containers. Therefore, create a file called `Dockerfile` (no ending) in the shared folder, with the following content:

```
FROM java:openjdk-7-jre
COPY /oasp4j-sample-server.war ../oasp4j-sample-server.war
EXPOSE 8080
```

This will use a java 7 image as a base, which gets downloaded from the public Docker repository and copy the application in the root folder of the VM. When the container is started, Docker exposes the port 8080 towards the VM. Now, go back to the Quickstart Terminal. In the shared folder, you can type `docker build -t java7 .` (Don't forget the full stop) to build an image with the above `Dockerfile`. You can now run the container with `docker run -v /dev/urandom:/dev/random -p 8080:8080 java7 java -jar oasp-sample-server.war`. The `-p 8080:8080` maps the port of the `boot2docker` to the port 8080 of the VM. The command `-v /dev/urandom:/dev/random` specifies where to get random numbers. Utilizing `urandom` drastically speeds up the boot process of a container in a docker environment. When the application is started, you can connect with the IP of the VM. You can find out the IP of the VM by typing `docker-machine ip` in a console of your host machine.

In the above case, IP is 192.168.99.100, so the sample app can be accessed in a browser of the Windows host with 192.168.99.100:8080. To list all running containers types, use `docker ps` command. In order to stop a running container, one can use `docker stop nameOfContainer`.

### 3.16.2 Fabric8io plugin

Various maven plugins enable the dockerization within the process of building the application. In the devon sample server, this is shown using the [fabric8io/docker-maven-plugin](#), integrated as an extra profile called `docker`. Executing `mvn -Pdocker install`

- Executes the build process,
- Grabs the bootable war file,
- Builds an image based on the configuration in the `server/pom.xml` and the `server/src/main/docker/assembly.xml`,
- Deploys the image depending on the environment variables e.g. To a local instance of docker,
- Starts the application, test if it is reachable through an URL and finally
- Stops the container.

In detail, if you want to test this on your local machine, you need to have docker installed as mentioned above. In the console, type `docker-machine env` to see the IP and other properties of your VM. To set the environment variables, so that the plugin deploys the application into your local docker instance, type the last line of that output (something like `eval $(C:\Program Files\ Docker Toolbox\docker-machine.exe env)`). Now, navigate to your workspace, where the project is located. In the `devon/sample` folder, you can now execute `mvn -Pdocker install` and see how the process starts. If you only want to build the container, you can navigate into `/devon/sample/server` and execute `mvn -Pdocker package docker:build` or start the container with `mvn -Pdocker docker:start`.

### 3.16.3 Docker tips

[[cookbook-dockerization\_run-war file on tomcat]] === Run WAR file on Tomcat

If you have a WAR file and want to run on Tomcat without installing you can use docker and mount the WAR file under the `/usr/local/tomcat/webapps/yourapp.war`.

```
docker run --name tomcat -it -p 8080:8080 --rm -v c:/path_to_your/app.war:/usr/local/tomcat/webapps/app.war tomcat:8-alpine
```

The format for Docker for Windows is `c:/` whereas if using Docker Toolbox with Virtualbox you should use `/c/Users/`

## 3.17 Production Line for Beginners (Practical Guide)

[[devon-guide-production-line-for-beginners\_1.-production\_line\_description]] == 1. Production Line description The Production Line is a set of tools that will allow us to trigger some actions like testing, packaging, building, storage, etc. This will help us to improve our development having a quick and real-time feedback about how good is our work according to some pre-set rules.

### 3.17.1 2. Jenkins

Jenkins will allow us to create those processes that will evaluate, test, package, build or whatever action we might need. Ideally, those processes should be triggered every time we push some changes to the repository of the project we want to pass through the Production Line.

#### 2.1 Jobs

A Jenkins job will execute a single and specific action. For example, we want to ng build an Angular2 project. We could do it by creating a Jenkins job that fetches the repository and builds what's on it.



The way of creating it consists in add the URL of the location of the repository we're working on. In this case is the Production Line's Gerrit, but it could equally be Git or SVN (necessary little bit of further work with SVN, but valid as well).

#### Source Code Management

None

Git

Repositories

Repository URL

<https://devon.s2-eu.capgemini.com/gerrit/p/devonfw-angular2-k>

Credentials

jenkins (Jenkins Slave Authentication)

Then, add to our Job all the tools we need to use the commands (necessary to have them previously installed in our instance either as a general tool or as a custom tool), in that case we'd need NodeJS and Angular-CLI:

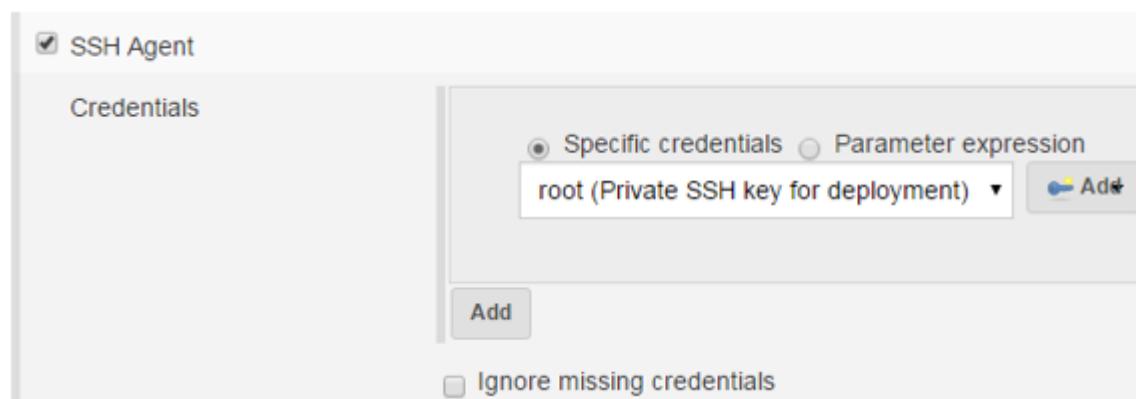


The screenshot shows the Jenkins 'Tools to install' configuration page. It displays two sections for tool selection:

- Tool selection:** Node 6
- Tool selection:** Angular CLI

Below these sections is a button labeled "Add Tool".

Finally, it's time to set up our script, which is the main part of the Jenkins Job. In that script we tell our Job to, for example, install (via NodeJS) all the dependencies of the project on the PL's workspace, build the project (via Angular-CLI) as a distributable set of files and at the end, serve it to an external server (for example, why not?) via ssh connection, and perform some Docker actions in there (don't worry about that). Obviously, we'll need to tell the Job about this SSH thing, adding some credentials for it.



The screenshot shows the Jenkins 'SSH Agent' configuration page. It includes the following elements:

- SSH Agent:** A checked checkbox.
- Credentials:** A dropdown menu with two options:
  - Specific credentials (selected)
  - Parameter expression
- root (Private SSH key for deployment):** The selected credential entry.
- Add:** A button to add a new credential.
- Ignore missing credentials:** A checkbox.

## Build

### Execute shell

```
Command  npm install --silent
          ng build

          # Copy resulting "dist" folder from workspace to deployment server
          scp -o StrictHostKeyChecking=no -r dist root@EXTERNALSERVERIP:/root/temp/

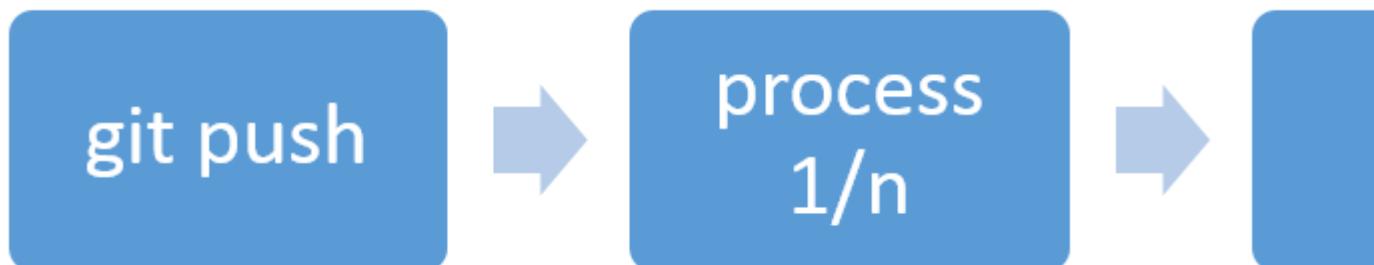
          # Launch application in Docker container
          ssh -o StrictHostKeyChecking=no root@EXTERNALSERVERIP << EOF
          docker rm -f angular2
          docker run -itd --name=angular2 -p 80:80 nginx
          docker exec angular2 bash -c "rm /usr/share/nginx/html/*"
          docker cp temp/dist/. angular2:/usr/share/nginx/html/
          EOF
```

See [the list of available environment variables](#)

If everything is correctly set up, we should build our Jenkins Job and wait for it to give us some feedback.

## 2.2 Pipelines

Let's think about pipelines as a set of jobs. Using pipelines will give us good feedback about each process of the chain as well.



A pipeline will mostly consist in a Jenkins script (written in Groovy) that will define every single task or process that we want to trigger for our project. We'll define each process as a stage and one could be used, for example, to trigger project's tests, another one to build or package the project, and another one to deploy this built project to somewhere.

## Pipeline

[Definition](#)[Pipeline script](#)

```
Script
1  node {
2
3      stage "Process 1/n"
4      |    sh "p1"
5      stage "Process 2/n"
6      |    sh "p2"
7
8      (...) X
9
10     stage "Process n/n"
11     |    sh "pn"
12
13 }
```

 Use Groovy Sandbox[Pipeline Syntax](#)

There is a helpful guide to write some pipeline commands. Just press "Pipeline Syntax", below the script box.

### 3.17.2 3. Nexus

[[devon-guide-production-line-for-beginners\_3.1-upload artifacts to a nexus repository]] === 3.1 Upload artifacts to a Nexus repository

This process seems quite interesting as a final stage of our pipeline's script. It makes sense to trigger project's tests, then build and package the application and then publish it to a repository where someone could download it and use it, or just have it as a backup. This process will be defined from our project's pom.xml file. There is a section called `<distributionManagement>` that will tell Maven where to upload the created project's artifact.

```
<distributionManagement>
  <repository>
    <id>devon.releases</id>
    <name>devon Releases</name>
    <url>https://yourPLinstanceIP/nexus/content/repositories/releases</url>
  </repository>
  <snapshotRepository>
    <id>devon.snapshots</id>
    <name>devon Snapshots</name>
    <url>https://yourPLinstanceIP/nexus/content/repositories/snapshots</url>
  </snapshotRepository>
</distributionManagement>
```

Having this, we'll store two artifacts on the Nexus repositories every time we package our project on a Production Line job or pipeline. In case of this pipeline, we build an Angular2 project as a JS/HTML project. After we store it on Nexus, we can always download it and publish it wherever we want.

Releases hosted ANALYZE maven2 Release In Service

Snapshots hosted ANALYZE maven2 Snapshot In Service

## Snapshots

**Browse Index** Browse Storage Configuration Routing Summary

Refresh

- + [cobigen-core-api](#)
- + [cobigen-core-parent](#)
- + [cobigen-core-test](#)
- + [cobigen-javaplugin](#)
- + [cobigen-jsonplugin](#)
- + [cobigen-master-parent](#)
- + [cobigen-maven-plugin](#)
- + [cobigen-parent](#)
- + [cobigen-propertyplugin](#)
- + [cobigen-senchaplugin](#)
- + [cobigen-tempeng-freemarker](#)
- + [cobigen-textmerger](#)
- + [cobigen-xmlplugin](#)
- [devonfw](#)
  - + [dev](#)
  - [devonfw-angular2](#)
    - [1.0-SNAPSHOT](#)
      - [devonfw-angular2-1.0-SNAPSHOT-resources.zip](#)
      - [devonfw-angular2-1.0-SNAPSHOT.jar](#)
  - + [devonfw-bom](#)
  - + [devonfw-sample](#)
  - + [devonfw-sample-core](#)
  - + [devonfw-sample-server](#)
  - + [extjs-sample](#)
  - + [modules](#)
- + [io](#)

[[devon-guide-production-line-for-beginners\_3.2-upload files to a nexus repository (direct upload)]] ===  
3.2 Upload files to a Nexus repository (direct upload)

Surely there will be some cases on we will need to store something on a Nexus repository we've created before. For example, for Sencha projects, we need to retrieve the license if we want to compile or build the project. We will follow the next steps:

- 1. Get the project from our repository (Gerrit/GitLab/GitHub...)
- 2. Get the license from Nexus (ext.zip)
- 3. Follow with every extJS command we need
- 4. ...

So, the question is: How we upload a file on a Nexus repository of the PL? There is an API to deal with it:

```
curl -L -u myusername:mypassword --upload-file ext.zip https://url-to-your-nexus-repository
```

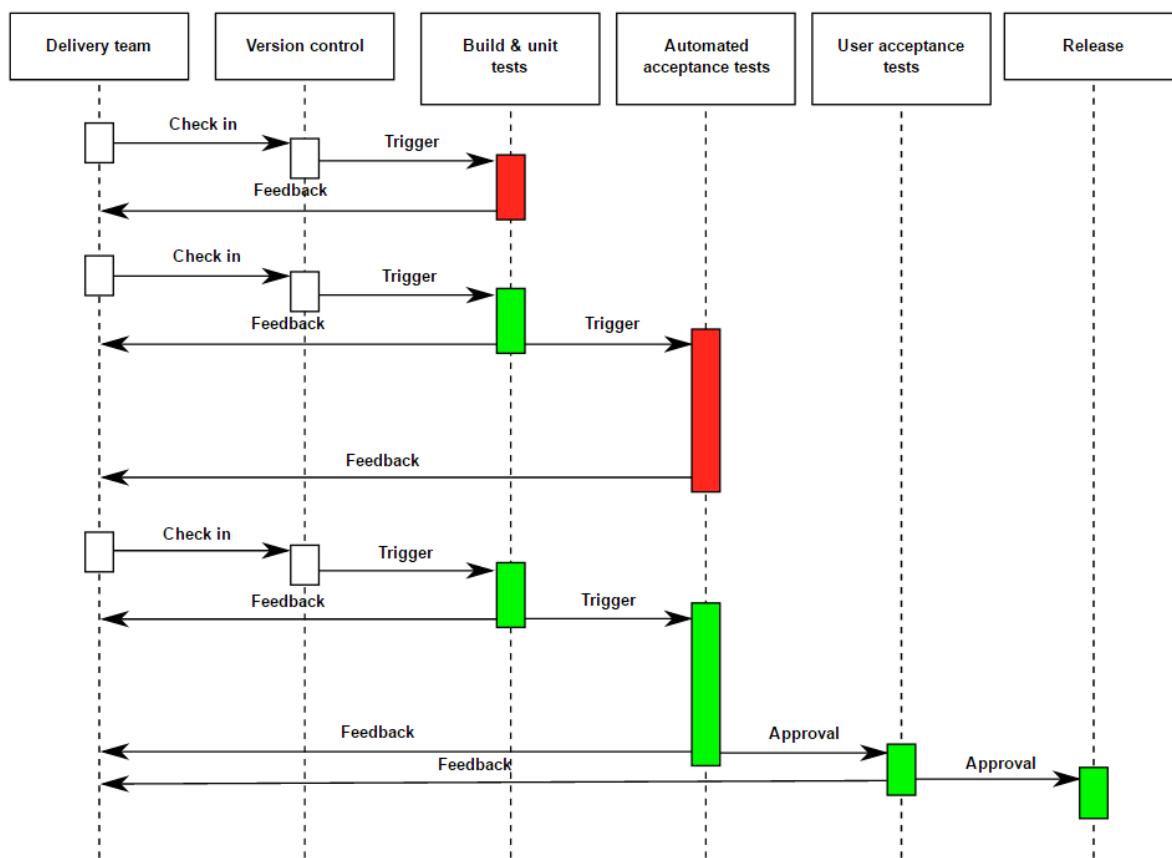
This command below will upload the file ext.zip on the nexus repository with the url <http://url-to-your-nexus-repository> using the credentials myusername/mypassword.

## 3.18 Production Line

### 3.18.1 Introduction

The *continuous delivery* is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently. The approach helps to reduce the cost, time, and risk of delivering the changes, by allowing more incremental updates into the applications in production.

Continuous delivery treats the notion of a *deployment pipeline* based on a *mistake-proofing* approach: a set of validations through which a piece of software must pass on its way to release. Code is compiled if necessary and then packaged by a build server every time a change is committed to a source control repository, then tested by a number of different techniques (possibly including manual testing) before it can be marked as releasable.



[[devon-guide-production-line\_continuous-delivery benefits]] == Continuous Delivery benefits The practices at the heart of continuous delivery help us achieve several important benefits:

- **Low risk releases.** The primary goal of continuous delivery is to make software deployments painless, low-risk events that can be performed at any time, on demand. With *continuous delivery* can be relatively straightforward to achieve zero-downtime deployments that are undetectable to users.
- **Faster time to market.** It's not uncommon for the integration and the test/fix phase of the traditional phased software delivery lifecycle to consume weeks or even months. When teams work together to automate the build and deployment, environment provisioning, and regression testing process,

developers can incorporate integration and regression testing into their daily work and these phases will be completely removed. It is always preferred to avoid the large amounts of re-work that plague the phased approach.

- **Higher quality.** When developers have automated tools that discover regressions within minutes, teams are free to focus on their efforts on user research and higher level testing.
- **Lower costs.** By investing in build, test, deployment and environment automation, the cost of making and delivering incremental changes to software is substantially reduced. Moreover, many of the fixed costs associated with the release process are eliminated.
- **Better products.** Continuous delivery makes possible to get feedback from users throughout the delivery lifecycle based on working software. This means we can avoid the 2/3 of the features we build that deliver zero or negative value to our businesses.
- **Happier teams.** By removing the low-value painful activities associated with software delivery, we can focus on what we care about most—continuously delighting our users.

[[devon-guide-production-line\_the-production line]] == The Production Line The Production Line is the set of methods and tools to facilitate the implementation of the *continuous delivery* methodology in a Devon project, covering all the phases involved in the application development cycle from requirements to testing and hand-off to the client.

Created to make the easier inclusion of all the *continuous delivery* tools in a project, the Production Line will be automated and provisioned within few hours from the moment a project starts, providing access to the set of tools over the Production Line interface on <http://devon.s2-eu.capgemini.com>.

After logging in, the tools can be accessed over a drop down menu (called *Services*) in the top menu bar.

## Prerequisites

To implement the Production Line in your project, you only need:

- A Production Line instance.
- A Remote Linux host for deployment.

[[devon-guide-production-line\_devonfw-continuous delivery infrastructure]] == Devonfw Continuous Delivery infrastructure

## Tools

### Git

[Git](#) is a version control system, that helps a software team to manage the changes in a source code over time. Version control software keeps the track of every modification in the code. It allows restoring the state to the earlier versions of the code or working on the parallel features of the software using branches.

### Gerrit

[Gerrit](#) is a code collaboration tool. It hosts the Git repository and extends available functionality. It implements the voting protocol, allowing automated code review by software tools as well as manual acceptance by a reviewer.

Gerrit can be placed in between the repository and the user's code push request to provide the ability to discuss a change before submitting.

### **Jenkins**

[Jenkins](#) is an automation engine with a great plugin ecosystem to support the majority of the tools surrounding *continuous integration*, *automated testing* or *continuous delivery*. It provides tools for scheduling and automating the whole build process for the Devonfw apps managing the trigger and build processes.

### **SonarQube**

[SonarQube](#) is a tool for continuous inspection of code quality, preventing redundancies, complexity and aiming to approach to the code conventions and good practices. It performs static code analysis and allows gathering reports of the various tests performed on the application. Provides a single point with web GUI, where developers can check the test results.

### **Maven**

[Maven](#) is a build automation tool used primarily for the Java projects. Was originally created to achieve a clear definition of how to build an ANT project. Over the time, thanks to the community support and its plugin system, it evolved into a fully functional JAVA project management system. Within the *continuous integration*, the build process of the Devonfw applications is executed through Maven and only initiated by Jenkins.

### **Nexus**

[Nexus](#) is a repository providing centralized storage place for the JAVA artifacts – JAR / WAR files containing built applications.

### **Tomcat**

[Tomcat](#) is an open-source Java Servlet Container that implements several Java EE specifications, including Java Servlet, JavaServer Pages (JSP), Java EL, and WebSocket, and provides a "pure Java" HTTP web server environment in which Java code can run.

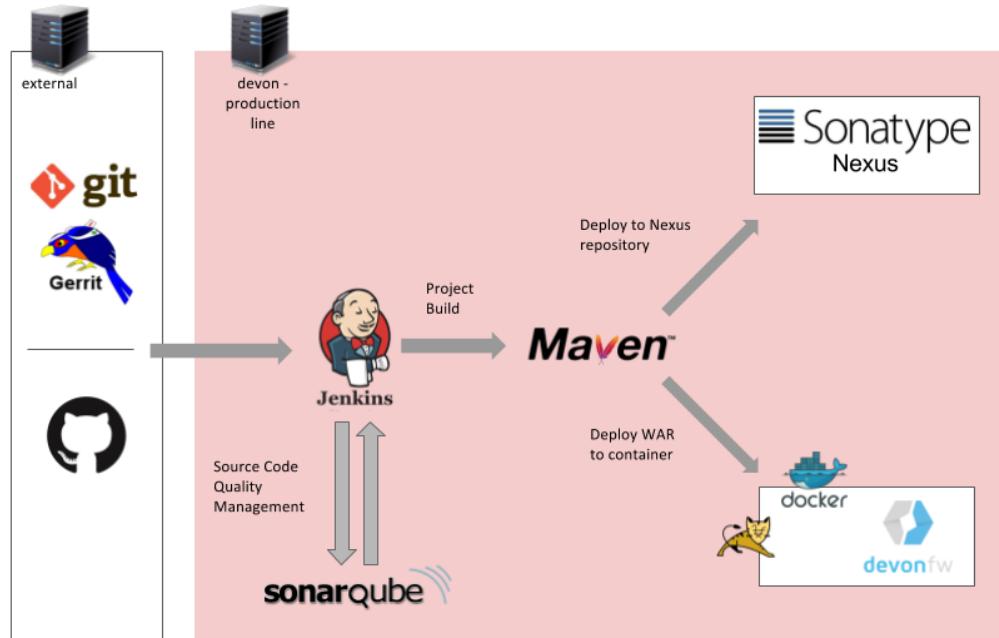
### **Docker**

[Docker](#) is a lightweight virtualization software allowing wrapping the applications into containers – running images with all the prerequisites and dependencies needed for the application to run. By letting go of the operating system burden, through the usage of the underlying host operating system, Docker containers can be started almost instantly. Additionally, Docker provides a set of tools that support management of the containers, hosting image repositories and many others.

### **Schema**

The *continuous delivery* concept is applied in the context of Devonfw apps with Jenkins as the core of the process and the rest of tools surrounding it.

The following schema shows the infrastructure of the tools used for the Devonfw Continuous Integration and their relations.



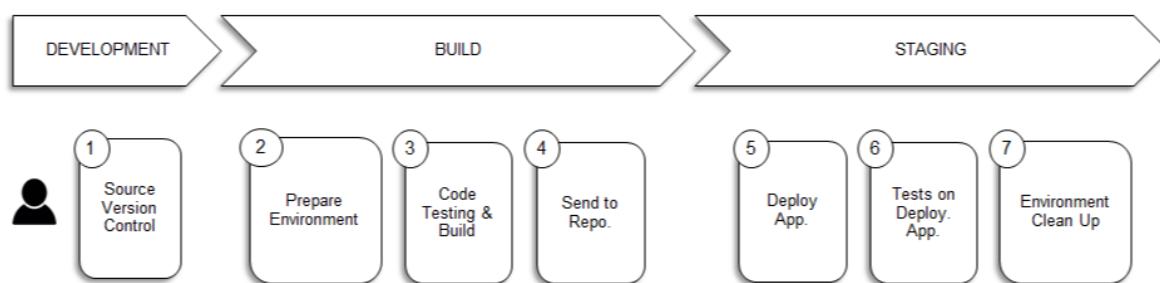
- A change in the project's git repository is registered (commit, push).
- Jenkins, as we just mentioned the core of *continuous integration*, gets triggered by that changes.
- Then, it builds and tests the project using *Maven*
- The resulting artifacts can be either deployed to a *Nexus* repository or to an app container (Docker, Tomcat).
- During the integration process a *SonarQube* instance manages the project's source quality.

If some of these stages fails or doesn't fit few requirements, all the process can be frozen until a solution is included in the content of the project. Once this happens, complete process will start again.

[[devon-guide-production-line\_production-line implementation for devonfw projects]] == Production Line implementation for Devonfw projects

[[devon-guide-production-line\_continuous-delivery pipeline]] == Continuous Delivery Pipeline While preparing the process of the automated build and testing, a good practice is to organize the development processes of the project in the form of the pipeline, that provides a clear view of its stages. This pipeline is reflected in Jenkins job stages and facilitates organization and issue identification.

Below, you can find the *continuous delivery* pipeline used in a basic Devonfw app:



1. Code commits into source version control tool, triggers the Jenkins job. Alternatively, it can be triggered manually.
2. The environment is prepared for the deployment – the prerequisites are checked and provisioned if not met.
3. Code is being built using Maven. During the build, the code checking tests are executed.
4. When the tests are finished successfully, the artifact and Docker ready image are sent to the repository, ready to be deployed in the staging environment.
5. When the environment is ready, Jenkins automatically deploys image from the repository.
6. After the application deployment, automatic tests are executed for the verification of the actual version on the test instance.
7. After the whole process, the environment is cleared, releasing hardware resources for the next run.

In terms of the tools, the previous schema could be represented as



Using the Pipeline plugin, it is possible to implement the Continuous Delivery pipeline as a Jenkinsfile, so the Jenkins job definition is treated as another piece of code checked into source control. The Jenkins jobs are each of the runnable tasks that are controlled or monitored by Jenkins.

This approach allows easy scalability and replicability of Jenkins implementation.

So, thanks to the Production Line the *continuous delivery* methodology can be included as part of the development of a Devonfw project achieving reliable releases, faster time to market, higher quality, lower costs and ultimately better products.

## 3.19 Devon in Bluemix

### 3.19.1 Introduction

[[devon-in-bluemix\_platform-as a service]] === Platform as a service

[Platform as a service \(PaaS\)](#) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.

#### Cloud Foundry

Cloud Foundry is an open source, multi cloud application platform as a service (PaaS) governed by the Cloud Foundry Foundation, a [501\(c\)\(6\) organization](#).

[Cloud Foundry](#) supports the full lifecycle, from initial development, through all testing stages, to deployment. It is therefore well-suited to the continuous delivery strategy. Users have access to one or more spaces, which typically correspond to a lifecycle stage. For example, an application ready for QA testing might be pushed (deployed) to its project's QA space. Different users can be restricted to different spaces with different access permissions in each.

#### Bluemix

[IBM Bluemix](#) is a cloud PaaS developed by IBM. It supports several programming languages and services as well as integrated DevOps to build, run, deploy and manage applications on the cloud. Bluemix is based on Cloud Foundry open technology and runs on SoftLayer infrastructure. Bluemix supports several programming languages, including Java, Node.js, Go, PHP, Swift, Python, Ruby Sinatra, Ruby on Rails and can be extended to support other languages such as Scala through the use of buildpacks.

### 3.19.2 Bluemix Services

The Services dashboard provides access to the Bluemix services available from IBM® and third-party providers. These include Watson, Internet of Things, Analytics, Mobile, and DevOps services. See more about Bluemix Services [here](#).

#### VCAP services

The VCAP\_SERVICES environment variable is a JSON object that contains information that you can use to interact with a service instance in Bluemix. The information includes service instance name, credential, and connection URL to the service instance. These values are populated into the VCAP\_SERVICES environment variable when your application is bound to a service instance in Bluemix.

The value of the VCAP\_SERVICES environment variable is available only when you bind a service instance to your application. You can view the application environment variables by using the following command:

```
cf env APP_NAME
```

[[devon-in-bluemix\_vcap-services on devon]] === VCAP services on Devon

A simple example below demonstrates how to work with VCAP services in Devon. In this scenario, a dashDB database service is configured in the Devon application. So, you can obtain the database credentials from VCAP service.

The JSON that Bluemix provides us with the VAP\_SERVICE variable is the next:

```
{  
  "dashDB": {  
    "name": "service-instance-name",  
    "label": "dashDB",  
    "plan": "Entry",  
    "credentials": {  
      "port": 50000,  
      "db": "BLUDB",  
      "username": "****",  
      "host": "23.246.206.254",  
      "https_url": "https://23.246.206.254:8443",  
      "hostname": "23.246.206.254",  
      "jdbcurl": "jdbc:db2://23.246.206.254:50000/BLUDB",  
      "ssljdbcurl": "jdbc:db2://23.246.206.254:50001/BLUDB:sslConnection=true;",  
      "uri": "db2://****:****@23.246.206.254:50000/BLUDB",  
      "password": "****",  
      "dsn": "DATABASE=BLUDB;HOSTNAME=23.246.206.254;PORT=50000;PROTOCOL=TCPIP;UID=****;PWD=****;",  
  
      "ssldsn": "DATABASE=BLUDB;HOSTNAME=23.246.206.254;PORT=50001;PROTOCOL=TCPIP;UID=****;PWD=****;Security=SSL;"  
    }  
  }  
}
```

### Note

You can find more details about dashDB service connection and VAP\_SERVICE [here](#).

Now, in the Devon application, you can simply create a configuration class and configure the DataSource:

```
@Configuration  
public class BeanJPAConfiguration {  
  public String username;  
  public String password;  
  public String url;  
  public String driverClassName = "com.ibm.db2.jcc.DB2Driver";  
  
  @Bean  
  @Primary  
  public DataSource dataSource() {  
    String VCAP_SERVICES = System.getenv("VCAP_SERVICES");  
  
    if (VCAP_SERVICES != null) {  
      setDataSourceProperties(VCAP_SERVICES);  
    }  
    return DataSourceBuilder.create()  
      .username(this.username).  
      password(this.password).  
      url(this.url).  
      driverClassName(this.driverClassName).  
      build();  
  }  
  
  public void setDataSourceProperties(String VCAP_SERVICES) {  
    JSONObject jsonObj = new JSONObject(VCAP_SERVICES);  
    JSONArray jsonArray;  
  
    if (jsonObj.has("dashDB")) {  
      jsonArray = jsonObj.getJSONArray("dashDB");  
      // Transform the JSONArray to JSONObject because JSONArray can't find by string key  
      jsonObj = jsonArray.toJSONObject(new JSONArray().put("dashDB"));  
      jsonObj = jsonObj.getJSONObject("dashDB");  
    }  
  
    if (jsonObj.has("credentials")) {  
      ...  
    }  
  }  
}
```

```
jsonObj = jsonObj.getJSONObject("credentials");
if (jsonObj != null) {
    this.username = jsonObj.getString("username");
    this.password = jsonObj.getString("password");
    if (jsonObj.has("jdbcurl")) {
        this.url = jsonObj.getString("jdbcurl");
    } else if (jsonObj.has("ssljdbcurl")) {
        this.url = jsonObj.getString("ssljdbcurl");
    }
}
```

As you can see, the check is applied to make sure that if VCAP\_SERVICES exist. So, if it doesn't exist, you can configure the other database, throw an error, etc.

This is a simple way to use Bluemix services, you can see [Spring Cloud Foundry](#) too.

### Note

In the example, you are learning how to obtain the credentials of the database. If you want to know how to configure a DB2/dashDB database, you can see more details [here](#).

[[devon-in-bluemix\_logs-in bluemix]] == Logs in Bluemix

### Devon logging

Devon uses [OASP logging module](#) as a logging system. The module uses [SLF4J](#) API and the [Logback](#) implementation and the OASP wiki contains an [excellent entry](#) explaining its configuration.

By default, the logging system uses the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Configuration file for logback -->
<configuration scan="true" scanPeriod="60 seconds">
    <property resource="io/oasp/logging/logback/application-logging.properties" />
    <property name="appname" value="restaurant"/>
    <property name="logPath" value="../logs"/>
    <include resource="io/oasp/logging/logback/appenders-file-all.xml" />
    <include resource="io/oasp/logging/logback/appender-console.xml" />

    <root level="DEBUG">
        <appender-ref ref="ERROR_APPENDER"/>
        <appender-ref ref="INFO_APPENDER"/>
        <appender-ref ref="DEBUG_APPENDER"/>
        <appender-ref ref="CONSOLE_APPENDER"/>
    </root>

    <!-- Minimize infrastructure debug logs -->
    <logger name="org.dozer" level="INFO"/>
    <logger name="org.flywaydb" level="INFO"/>
    <logger name="org.springframework" level="INFO"/>
    <logger name="org.hibernate" level="INFO"/>

</configuration>
```

In the above configuration, each log level is written in its own file.

### Note

Each appender has its own XML configuration file that can be found on: <https://github.com/oasp/oasp4j/tree/develop/modules/logging/src/main/resources/io/oasp/logging/logback>.

## Bluemix logging

Bluemix does not allow users to navigate through any log files and it uses its own tool to look into the logs. The Bluemix environment has a console to show the logs. Due to this fact, the default configuration for the Devon logging system is not appropriate for the environments like Bluemix.

To adapt the logging system, Devon users are required to make certain changes in the default configuration. The file appenders have no sense in this environment, so they must be removed. The following example, could be a valid configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Configuration file for logback -->
<configuration scan="true" scanPeriod="60 seconds">
  <property resource="io/casp/logging/logback/application-logging.properties" />
  <property name="appname" value="sample-bluemix-app"/>
  <include resource="io/casp/logging/logback/appender-console.xml" />

  <root level="DEBUG">
    <appender-ref ref="CONSOLE_APPENDER"/>
  </root>

  <!-- Minimize infrastructure debug logs -->
  <logger name="org.dozer" level="INFO"/>
  <logger name="org.flywaydb" level="INFO"/>
  <logger name="org.springframework" level="INFO"/>
  <logger name="org.hibernate" level="INFO"/>

</configuration>
```

The above configuration is intended to write all the logs equal or superior to DEBUG level in the Bluemix console.

## 3.20 Deployment on Wildfly

Following describes the steps to install and configure Wildfly 10.x

1. Download WildFly 10.x from <http://wildfly.org/downloads/>
2. Place the extracted folder under {DISTRIBUTION\_PATH}/software folder.
3. Run console.bat under {DISTRIBUTION\_PATH}.
4. Navigate to path {DISTRIBUTION\_PATH}/software/{extracted wildfly folder}/bin
5. Run standalone.bat file.

```
Administrator: C:\WINDOWS\system32\cmd.exe - standalone.bat
IDE environment has been initialized.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

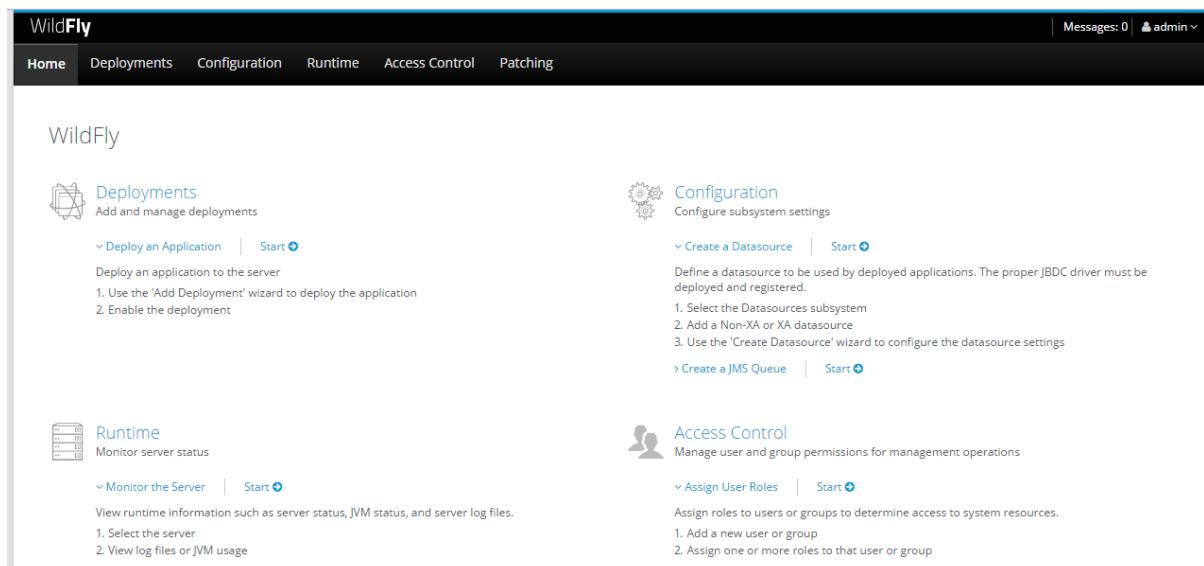
D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1>cd D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\wildfly-10.1.0\bin
D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\wildfly-10.1.0\bin>standalone.bat
Calling "D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\wildfly-10.1.0\bin\standalone.conf.bat"
Setting JAVA property to 'D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\java\bin\java'

=====
JBoss Bootstrap Environment
JBoss_HOME: "D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\wildfly-10.1.0"
JAVA: "D:\Devon_dist_2.1.1_new\Devon-dist_2.1.1\Devon-dist_2.1.1\software\java\bin\java"
JAVA_OPTS: "-Dprogram.name=standalone.bat -Xms64M -Xmx512M -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256M -Djava.net.preferIPv4Stack=true -Djboss.modules.system.pkgs=

=====
14:44:21,332 INFO [org.jboss.modules] (main) JBoss Modules version 1.5.2.Final
14:44:23,370 INFO [org.jboss.msc] (main) JBoss MSC version 1.2.6.Final
14:44:23,601 INFO [org.jboss.as] (MSC service thread 1-8) WFLYSRV0049: WildFly Full 10.1.0.Final (WildFly Core 2.2.0.Final) starting
```

It will start the administration console at <http://localhost:9990/console>. If the administration console asks for login credentials, provide "admin" as username and password.

Administration console is up and running at <http://localhost:9990/console/App.html#home>

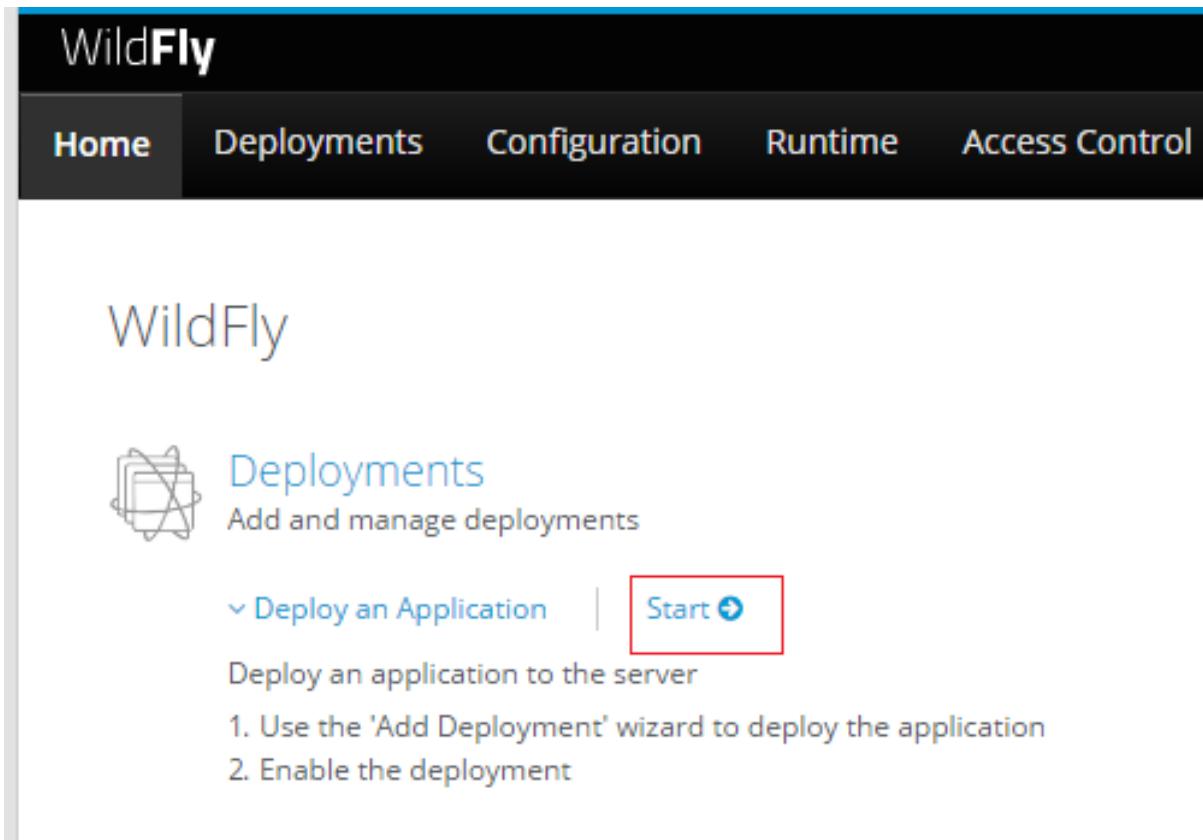


The screenshot shows the WildFly 10.1.0 administration console. The top navigation bar includes links for Home, Deployments, Configuration, Runtime, Access Control, and Patching. The Home tab is selected. The main content area is titled "WildFly" and contains five management sections:

- Deployments:** Add and manage deployments. Sub-options include "Deploy an Application" (with "Start" button) and "Monitor the Server" (with "Start" button). Instructions: 1. Use the 'Add Deployment' wizard to deploy the application. 2. Enable the deployment.
- Configuration:** Configure subsystem settings. Sub-options include "Create a Datasource" (with "Start" button) and "Create a JMS Queue" (with "Start" button). Instructions: 1. Select the Datasources subsystem. 2. Add a Non-XA or XA datasource. 3. Use the 'Create Datasource' wizard to configure the datasource settings.
- Runtime:** Monitor server status. Sub-options include "Monitor the Server" (with "Start" button). Instructions: 1. Select the server. 2. View log files or JVM usage.
- Access Control:** Manage user and group permissions for management operations. Sub-options include "Assign User Roles" (with "Start" button). Instructions: 1. Add a new user or group. 2. Assign one or more roles to that user or group.

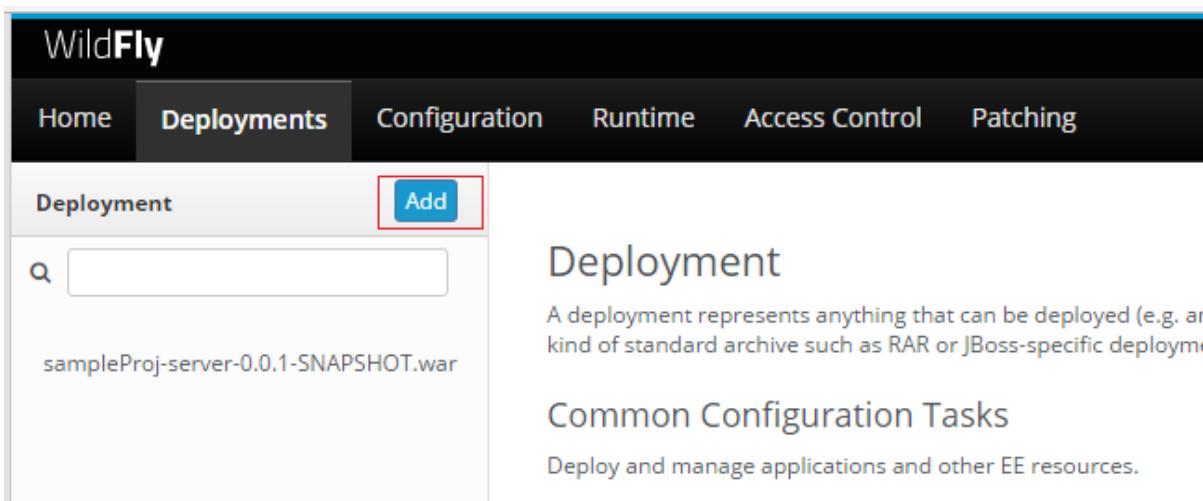
To run .war file on Wildfly administration console, follow below steps:

- Click on start (highlighted in below screenshot)



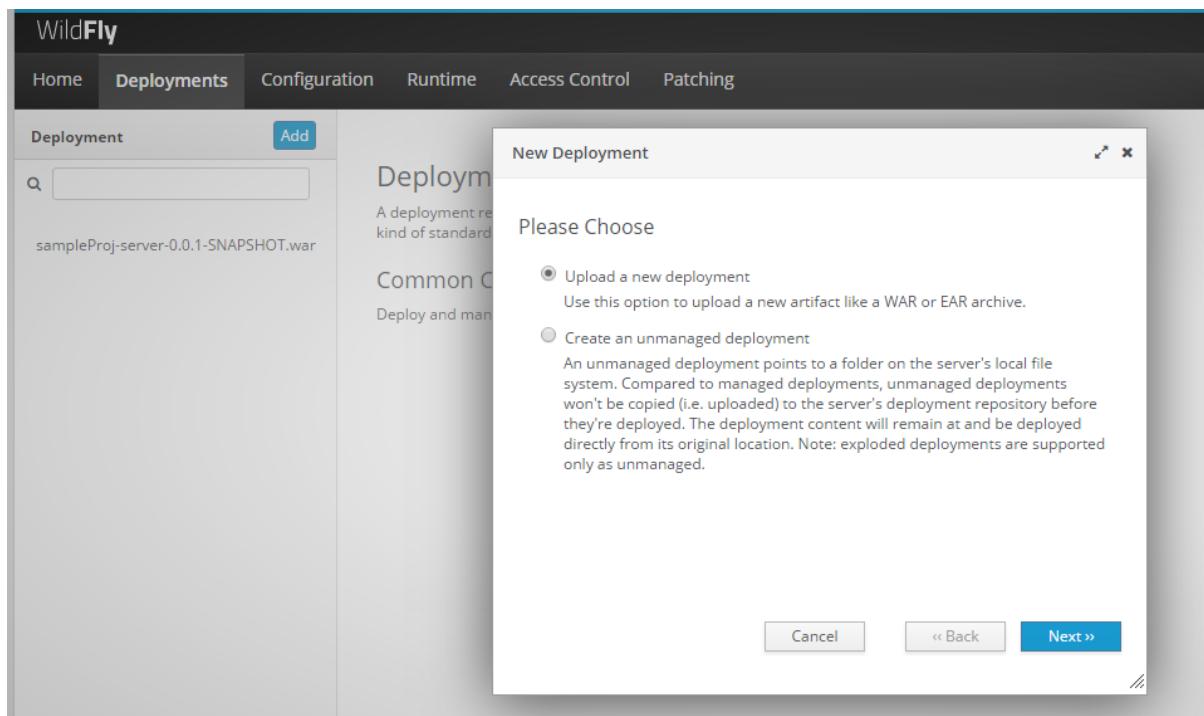
The screenshot shows the WildFly deployment management interface. The top navigation bar includes Home, Deployments, Configuration, Runtime, and Access Control. The Deployments tab is selected. Below the navigation, the title 'WildFly' is displayed, followed by 'Deployments' and 'Add and manage deployments'. A 'Deploy an Application' section contains a 'Start' button, which is highlighted with a red box. Below this, instructions state: 'Deploy an application to the server' and '1. Use the 'Add Deployment' wizard to deploy the application' followed by '2. Enable the deployment'.

- Click on Add button (highlighted in below screenshot)

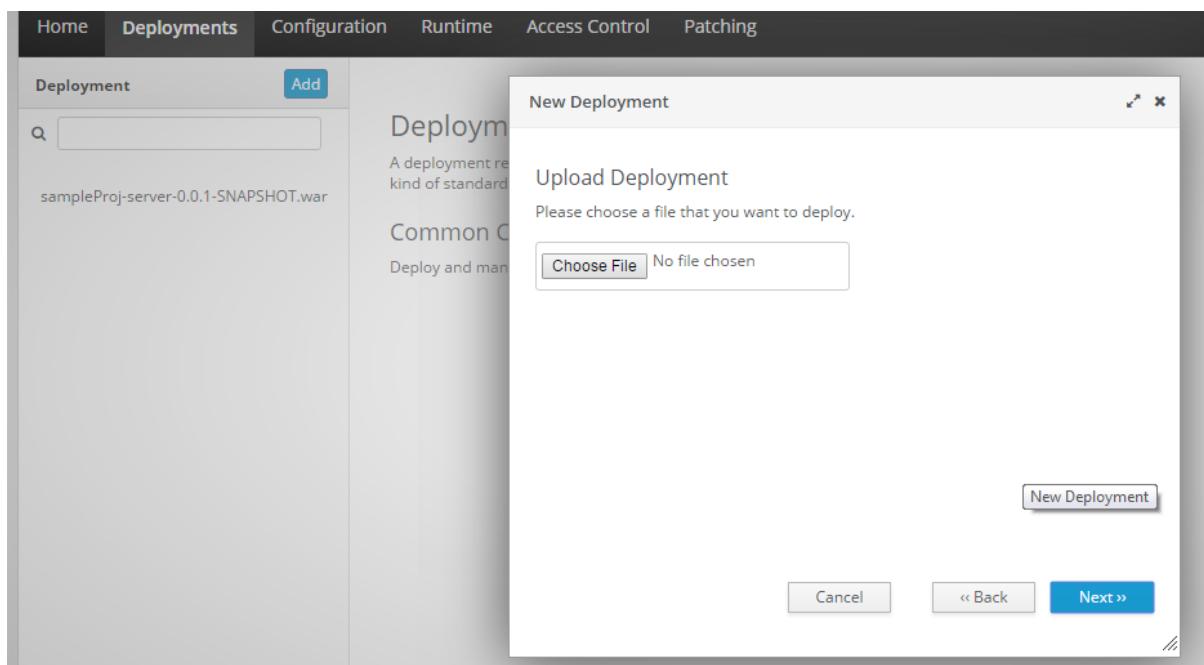


The screenshot shows the WildFly deployment management interface. The top navigation bar includes Home, Deployments, Configuration, Runtime, Access Control, and Patching. The Deployments tab is selected. On the left, a sidebar shows a 'Deployment' section with an 'Add' button, which is highlighted with a red box. The main content area is titled 'Deployment' and describes it as 'A deployment represents anything that can be deployed (e.g. a kind of standard archive such as RAR or JBoss-specific deployment)'. Below this, a 'Common Configuration Tasks' section is shown with the sub-instruction 'Deploy and manage applications and other EE resources.'

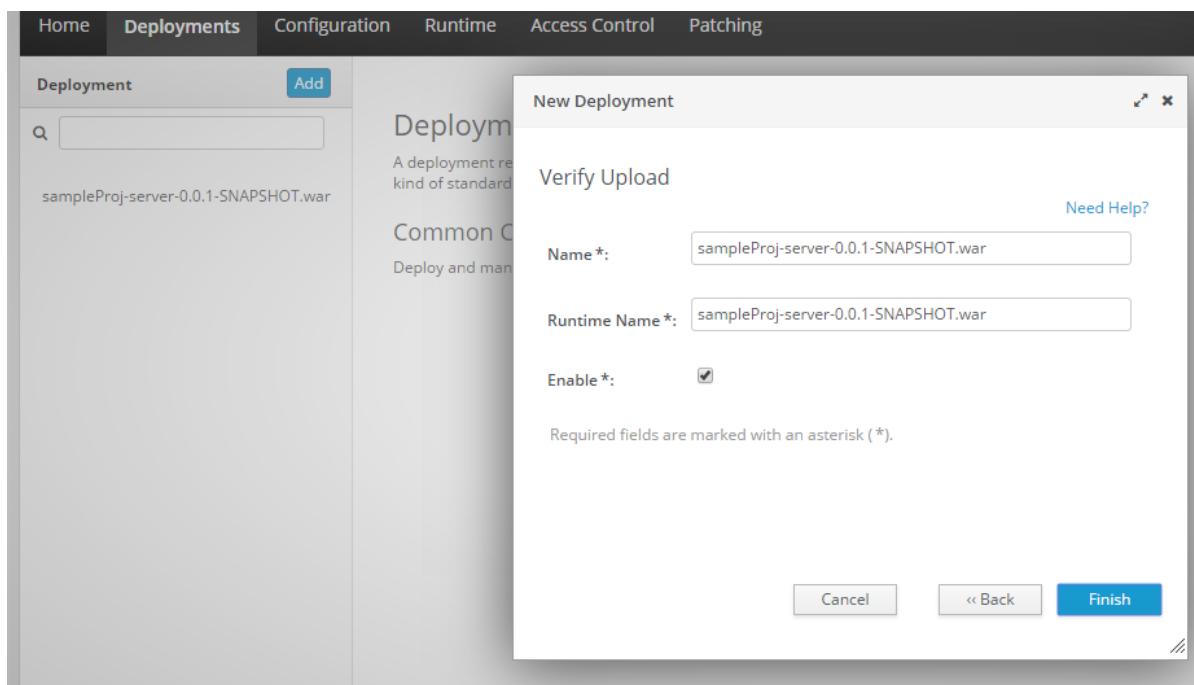
- Upload new deployment.



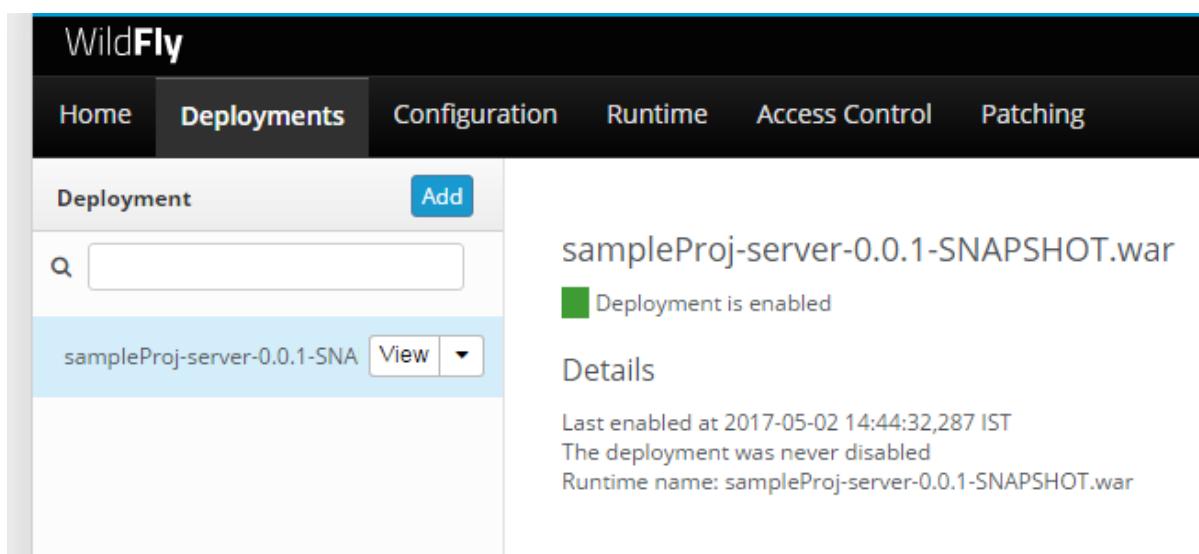
- Choose .war file for the deployment.



- Verify upload and finish



- Successful deployment



**Execute below mentioned steps to create .war file :**

- start a new oasp4j project from the template mvn -DarchetypeVersion=2.2.0 -DarchetypeGroupId=io.oasp.java.templates -DarchetypeArtifactId=oasp4j-template-server archetype:generate -DgroupId=io.oasp.application -DartifactId=sampleProj -Dversion=0.1-SNAPSHOT -Dpackage=io.oasp.application.sampleProj
- On SpringBootApp.java
  - Remove the exclude=xxxx option on the @SpringBootApplication annotation
  - Remove the @EnableGlobalMethodSecurity annotation
  - Remove the SpringBootBatchApp file.

- In pom.xml

Add below dependecies:

```
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-spring-boot-starter-jaxrs</artifactId>
<version>3.1.10</version>
</dependency>
<dependency>
<groupId>javax.xml.ws</groupId>
<artifactId>jaxws-api</artifactId>
<version>2.2.11</version>
</dependency>

<dependency>
<groupId>javax.xml.ws</groupId>
<artifactId>jaxws-api</artifactId>
<version>2.2.11</version>
</dependency>
```

Remove below dependency:

```
<!-- <dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-websocket</artifactId>
</dependency>
-->
<!--
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
-->
```

Modify below dependency:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
<exclusion>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-websocket</artifactId>
</exclusion>
</exclusions>
</dependency>
```

- create a file for a new rest controller

```
package io.oasp.application.sampleapp.general.service.impl.rest;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.stereotype.Component;

@Component
@Path("/hello")
public class HelloWorldEndpoint {
    @GET
    @Path("/world")
    @Produces(MediaType.TEXT_PLAIN)
    public String test() {
        return "Hello world!";
    }
}
```

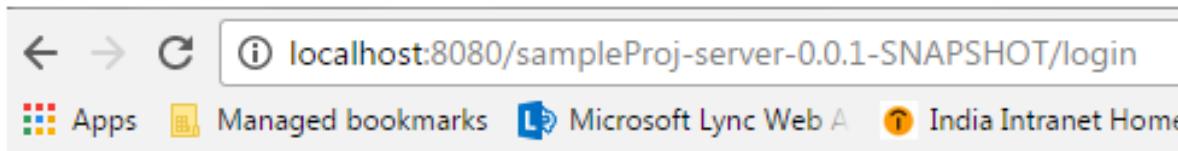
{}

- server logback.xml comment non console loggers:

```
<!-- <property name="logPath" value="../logs"/> -->
<!-- <include resource="io/oasp/logging/logback/appenders-file-all.xml" /> -->

<!-- <appender-ref ref="ERROR_APPENDER"/>
<appender-ref ref="INFO_APPENDER"/>
<appender-ref ref="DEBUG_APPENDER"/> -->
```

- Build your application with command "mvn clean install" and create the .war file and deploy it to wildfly 10 Administration console and start the application. You can hit the URL based on the context root of your project which is deployed on Wildfly.

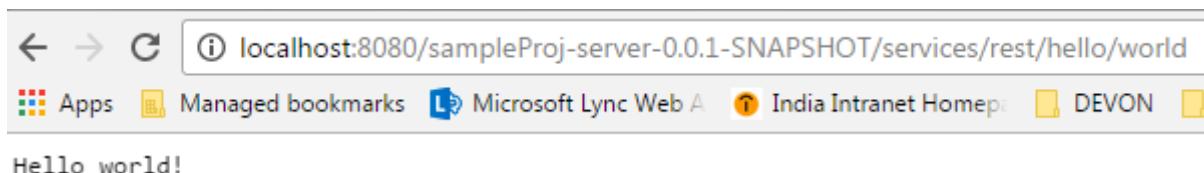


## Login with Username and Password

User:

Password:

- To test the webservice created on server, hit the URL <http://localhost:8080/sampleProj-server-0.0.1-SNAPSHOT/services/rest/hello/world> on browser.



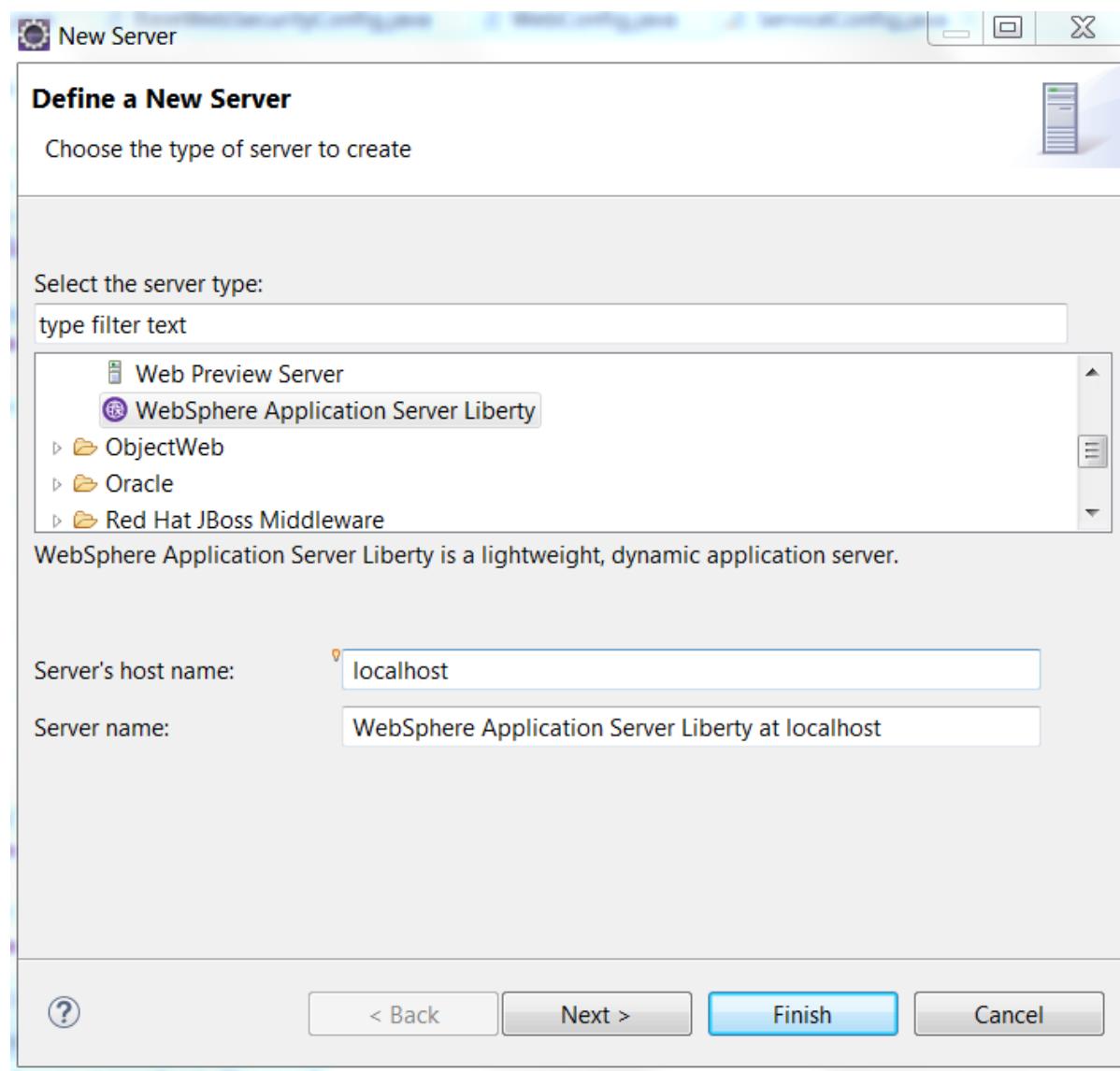
## 3.21 Deploy oasp4j application to WebSphere Liberty

Following describes the steps to install and configure WebSphere Liberty in Eclipse Neon that is provided as part of the Devonfw distribution and also details the steps to deploy oasp4j sample application created from oasp4j maven archetype template onto WebSphere Liberty.

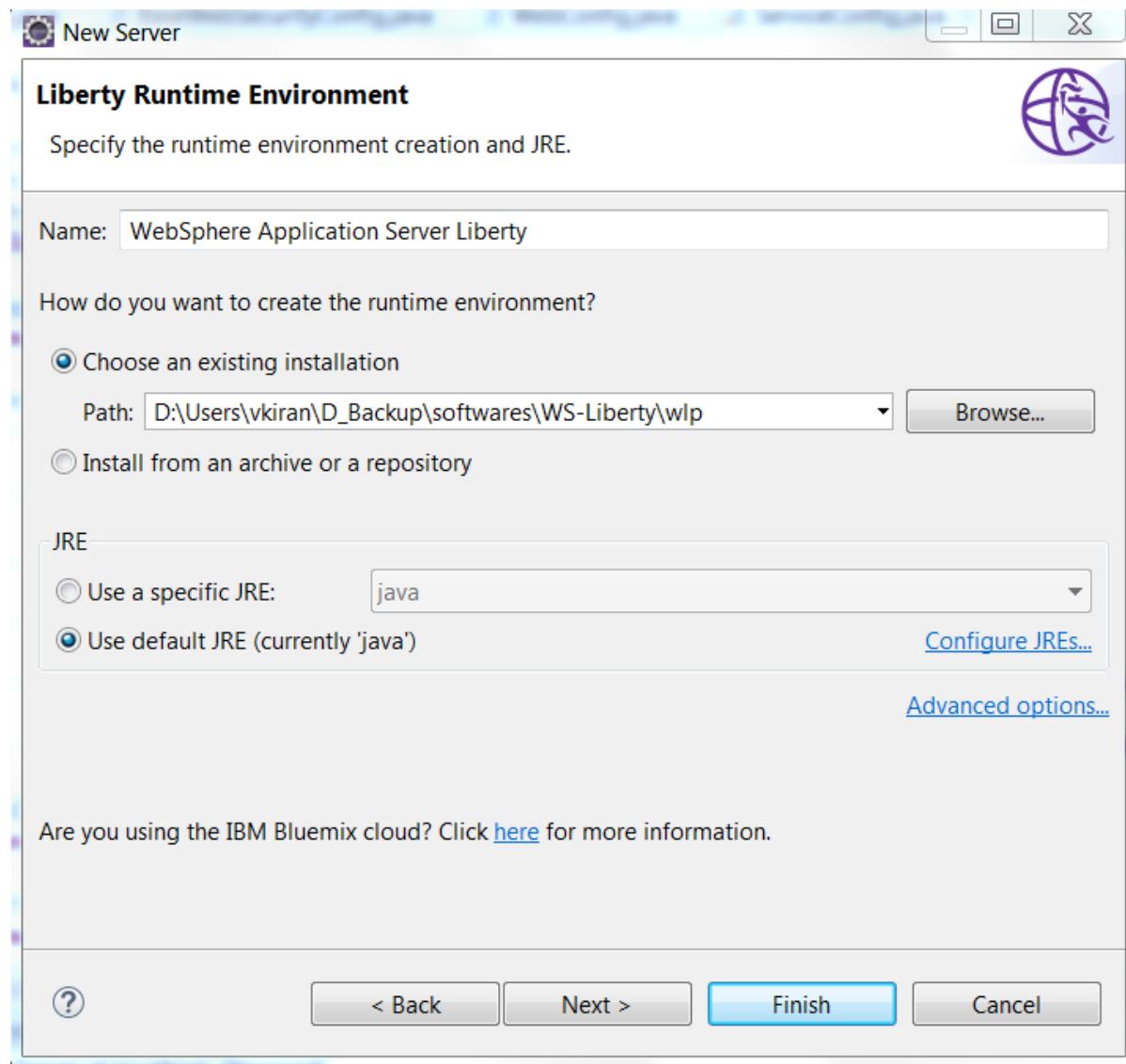
[[cookbook-deployment-on-websphere\_setup-and-configure\_websphere-liberty]] == Setup and Configure WebSphere Liberty

Launch Eclipse Neon from devonfw distribution and

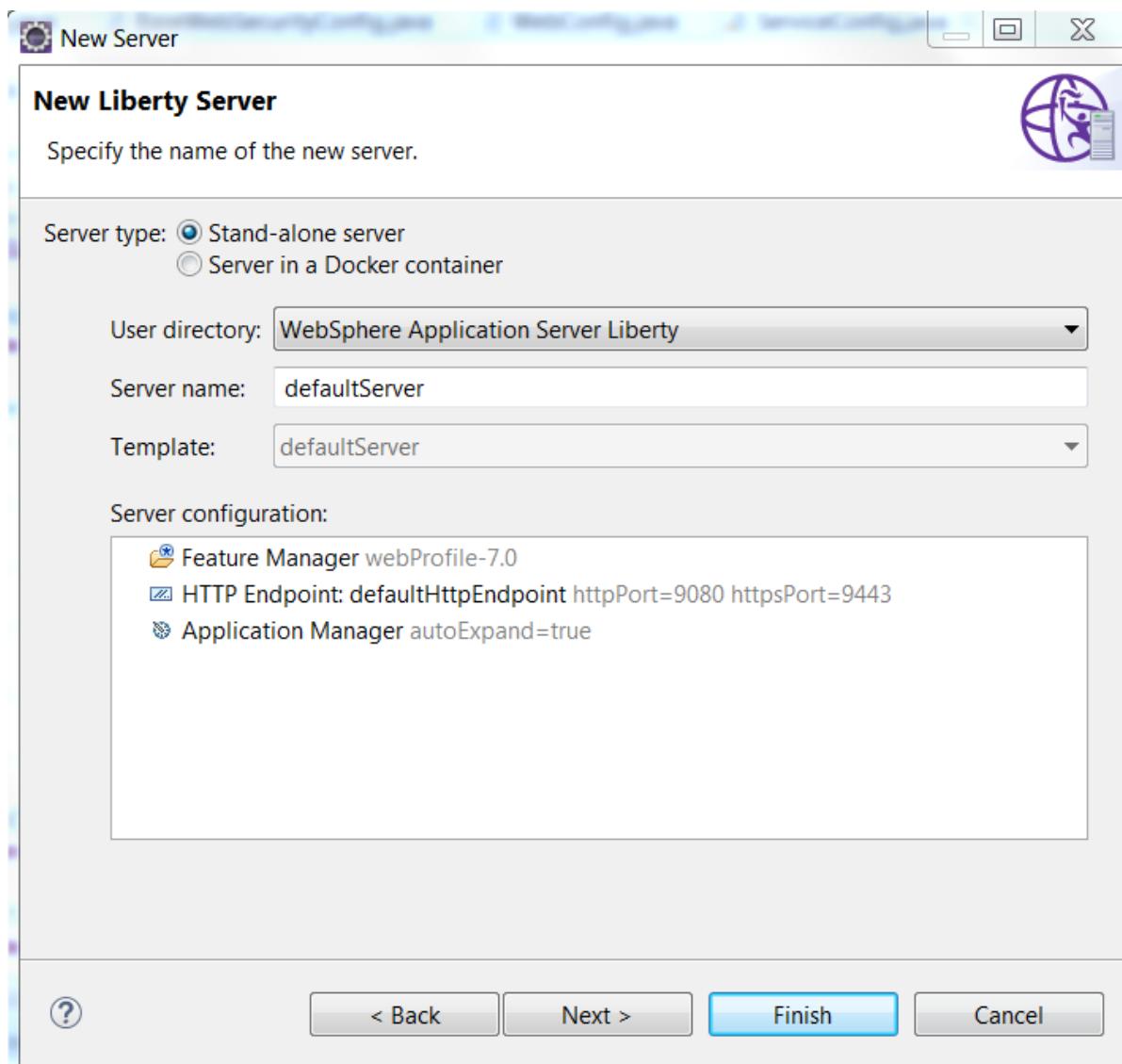
- Go to Help → Eclipse MarketPlace and search for 'WebSphere Liberty' and hit 'Enter'. In the results screen, select 'IBM WebSphere Application Server Liberty Developer Tools' and click 'install' button.
- To download Web sphere Liberty Profile, Go to [link](#) and click on the link 'Download Liberty (16.0.0.4) with Java EE 7 Web Profile'.
- To add new Websphere Liberty server, Go to 'Servers' View in eclipse and click on 'New' and in the screen that is displayed (see below), please select 'WebSphere Application Server Liberty' as the server type and click 'Next' button.



- In the screen that is displayed (see below), assuming that WebSphere Liberty is installed at 'D:\Users\vkiran\Backup\softwares\WS-Liberty\wlp' , click the radio button 'Choose an existing installation', and browse the Path 'D:\Users\vkiran\Backup\softwares\WS-Liberty\wlp' and click 'Next' button.



- In the screen that is displayed (see below), select the 'server type' as 'Stand-alone server' and enter server name in the text box 'Server name' and click 'Finish' button.



- Create a sample oasp4j application by referring the [link](#) . Alternatively, you can run the following command inside devonfw distribution.
- Assuming distribution is at D:\Devon2.1.0\Devon-dist\_2.1.0, run the command console.bat inside this directory and then run the following command

```
mvn -DarchetypeVersion=2.2.0 -DarchetypeGroupId=io.oasp.java.templates -DarchetypeArtifactId=oasp4j-template-server archetype:generate -DgroupId=io.oasp.application -DartifactId=libertyTest -Dversion=0.1-SNAPSHOT -Dpackage=io.oasp.application.libertyTest
```

- Add the following entries in the pom.xml of the server module just before the closing project tag

```
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-impl</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-core</artifactId>
```

```
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.apache.tomcat</groupId>
<artifactId>tomcat-jdbc</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-el</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>javax.ws.rs</groupId>
<artifactId>javax.ws.rs-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>xml-apis</groupId>
<artifactId>xml-apis</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.hibernate.javax.persistence</groupId>
<artifactId>hibernate-jpa-2.1-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>javax.annotation</groupId>
<artifactId>javax.annotation-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>javax.inject</groupId>
<artifactId>javax.inject</artifactId>
<scope>provided</scope>
</dependency>
</dependencies>
</dependencyManagement>
...

```

- In ...../general/service/impl/config/WebConfig.java, please remove the import statement 'import org.apache.catalina.filters.SetCharacterEncodingFilter;' and add the import 'import org.springframework.web.filter.CharacterEncodingFilter;'
- Code inside method setCharacterEncodingFilter in ...../general/service/impl/config/WebConfig.java should be changed accordingly to use org.apache.catalina.filters.SetCharacterEncodingFilter as follows :

```
.....
.....
CharacterEncodingFilter setCharacterEncodingFilter = new CharacterEncodingFilter();
setCharacterEncodingFilter.setEncoding("UTF-8");
setCharacterEncodingFilter.setForceEncoding(false);
.....
....
```

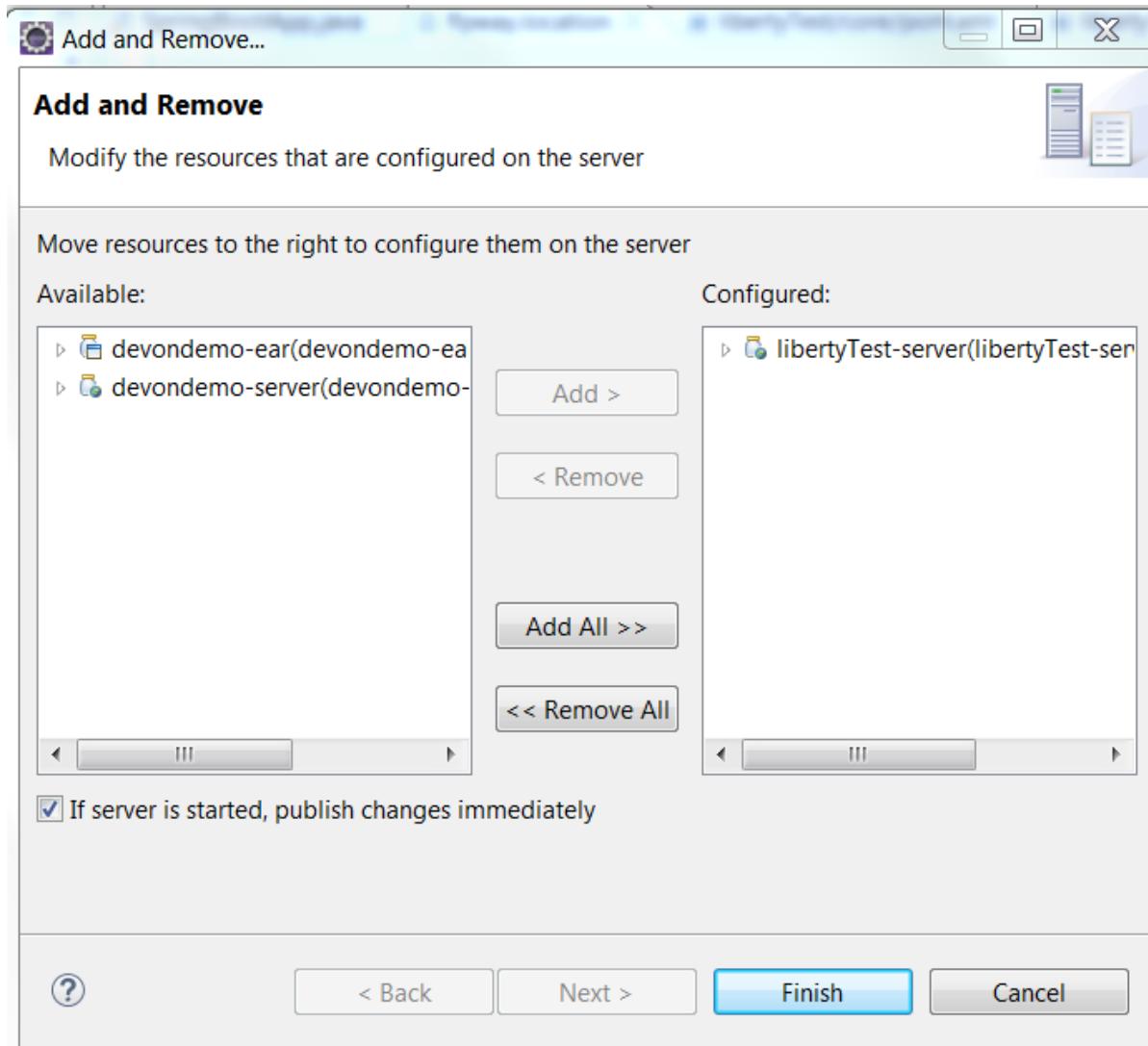
- Create an empty file flyway.location inside the directory core\src\main\resources\db\migration\
- Do 'mvn clean install' of the complete project
- Open server.xml of Websphere Liberty and add the following features,

```
...
```

```
<featureManager>
  <feature>webProfile-7.0</feature>
  <feature>localConnector-1.0</feature>
  <feature>jaxb-2.2</feature>
  <feature>jaxws-2.2</feature>
</featureManager>
```

```
...
```

- Deploy the war file on to the Websphere Liberty Profile and start the server.



- Once the application is published on to WebSphere Liberty, application url is logged in the Websphere console. Use this url and launch the application in browser.

## 3.22 Cobigen advanced use cases: SOAP and nested data

### 3.22.1 Introduction

In Devonfw we have a server-side code generator called Cobigen. Cobigen is capable of creating CRUD code from an entity or generate the content of the class that defines the user permissions. Cobigen is distributed in the Devonfw distribution as an Eclipse plugin, and is available to all Devonfw developers. If you want to go deeper in Cobigen you can visit the documentation of the [here](#). Previous versions of CobiGen is able to generate code for REST services only. Now it is possible to generate SOAP services code with it.

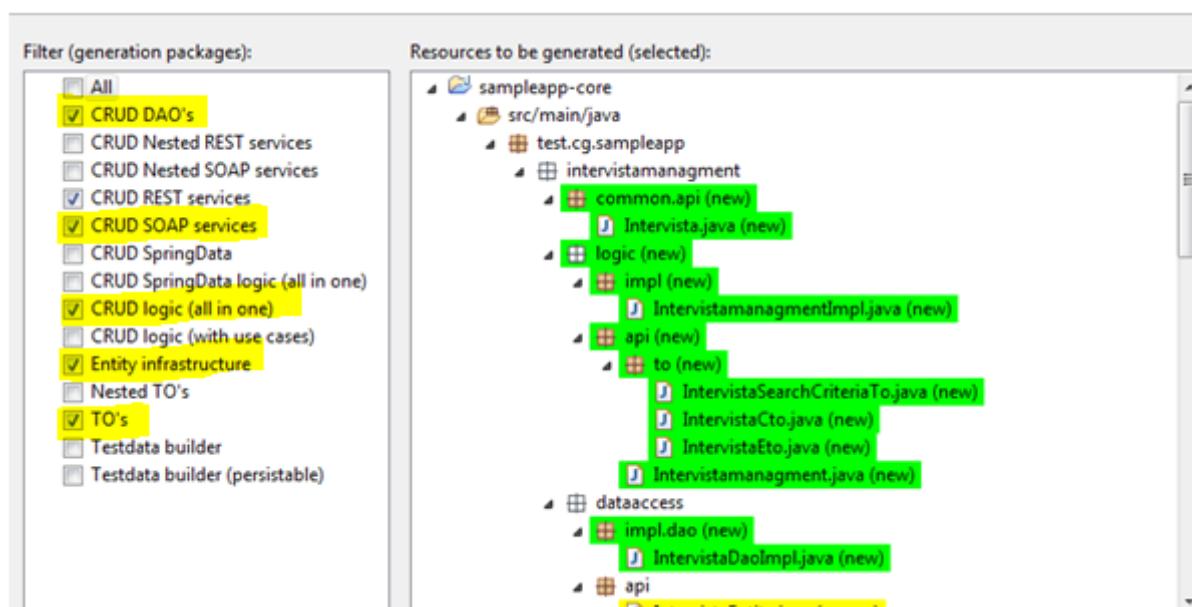
There are two usecases available in CobiGen:

1. CobiGen with SOAP functionality (without nested data)
2. CobiGen with SOAP functionality with nested data

[[cookbook-cobigen-advanced-use-cases-soap-and-nested-data\_cobigen-with\_soap\_functionality\_(without\_nested\_data)]] == CobiGen with SOAP functionality (without nested data)

To generate SOAP services with CobiGen follow below steps:

- If you are using cobigen for first time, setup CobiGen following instructions on <https://github.com/devonfw/devon-guide/wiki/getting-started-Cobigen#preparing-cobigen-for-first-use> .
- Clone latest code of CobiGen tool from <https://github.com/devonfw/tools-cobigen.git>
- Import \${workspace\_path}\tools-cobigen\cobigen-templates\templates-oasp4j code in eclipse.
- Once Cobigen is setup, we can start generating code.
- To generate SOAP code (without nested data) you need to select below options as shown in figure:



- Once above options are selected cobigen will generate code for SOAP services as well.

- To check if code is actually generated, check service package, it should have soap package inside it. Also in soap services which are generated will do CRUD operations on respective entity and will return ETO classes.

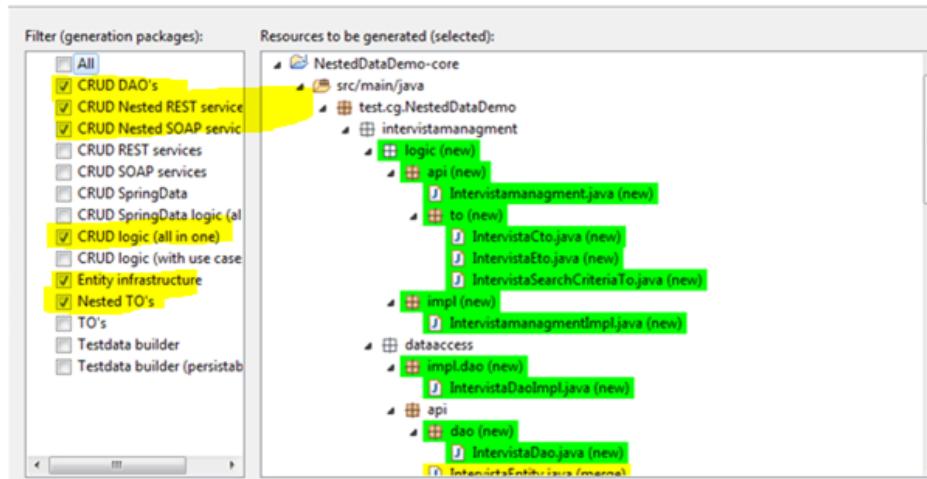
[[cookbook-cobigen-advanced-use-cases-soap-and-nested-data\_cobigen-with\_nested\_data]] ==  
CobiGen with nested data

### What is nested data?

Consider there are 3 or more entities which are interrelated with each other; we will generate a code which will return this relationship along with attributes in it i.e currently cobigen services return ETO classes, we have enhanced CobiGen to return CTO classes (ETO + relationship).

Steps to generate code with CobiGen are as below:

- If you are using cobigen for first time , setup cobigen following instructions on <https://github.com/devonfw/devon-guide/wiki/getting-started-Cobigen#preparing-cobigen-for-first-use> .
- Clone latest code of CobiGen tool from <https://github.com/devonfw/tools-cobigen.git>
- Import \${workspace\_path}\tools-cobigen\cobigen-templates\templates-oasp4j code in eclipse.
- Once Cobigen is setup, we can start generating code.
- To generate SOAP code with nested data you need to select below options as shown in figure:



Here we need to select option with nested prefix.

- Once above options are selected cobigen will generate code for SOAP services as well.
- To check if code is actually generated, check service package, it should have soap package inside it. Also in soap services which are generated will do CRUD operations on respective entity and you will have one method returning CTO class of that entity.

## 3.23 Compatibility guide for JAVA and TOMCAT

[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_what-this\_guide\_contains]] == What this guide contains

As, we have migrated to eclipse neon, which mandatorily uses Java8, if any project or user wants to use JAVA7, this guide will assist to do so. We also have integrated *TOMCAT8* with the Devon distribution and oasp4j IDE. Therefore, it also describes the steps required to use *Tomcat7* with the distributions.

### 3.23.1 Using JAVA7

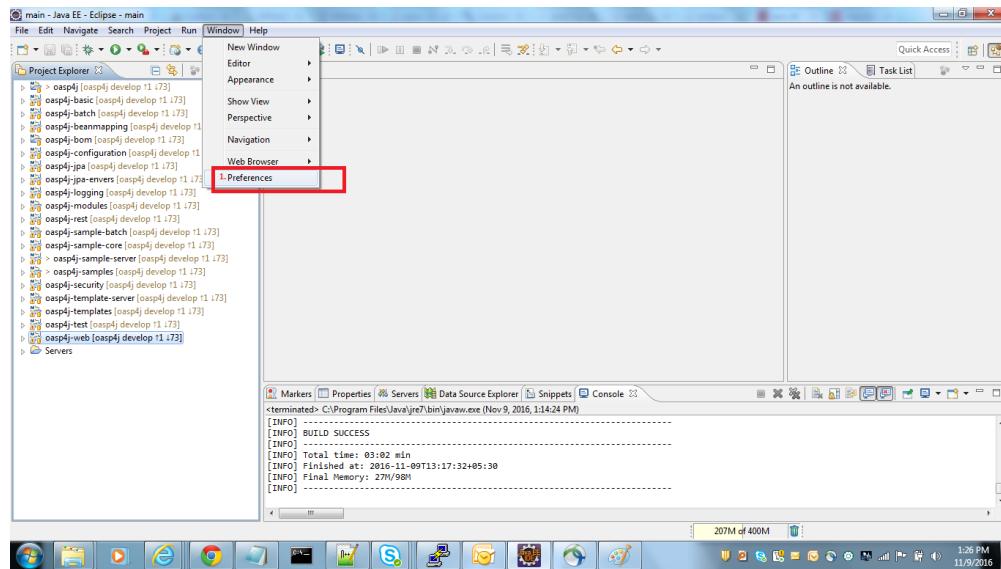
#### Download JAVA7

One can download java7 from [here](#). Once java7 is downloaded, run .exe and follow instructions.

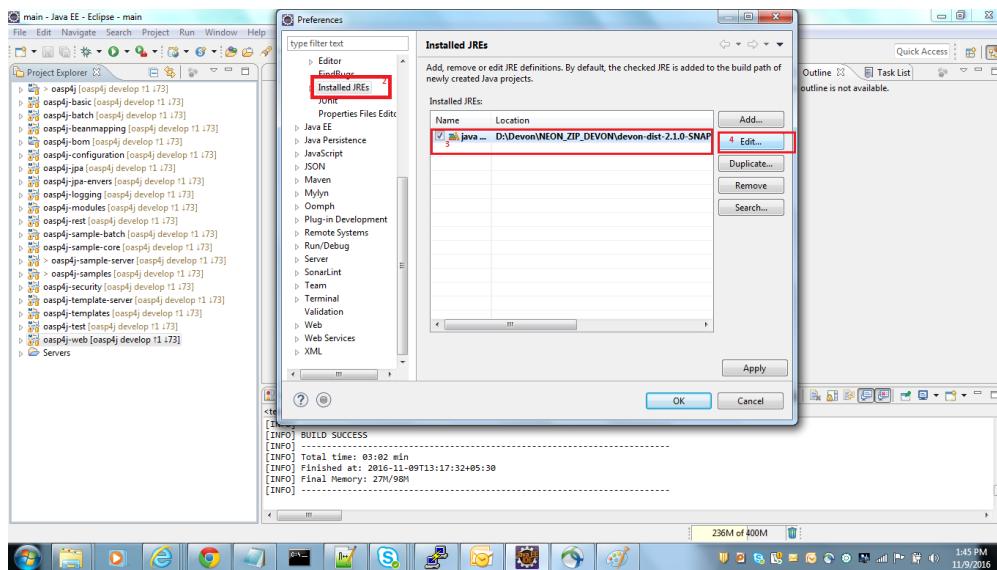
[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_change-installed\_jre\_in\_eclipse]] === Change installed jre in eclipse

When you open the eclipse, follow the below steps to change installed jre preferences:

[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_1.-go\_to\_preferences]] ===== 1. Go to Preferences



[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_2.-go\_to\_installed\_jres]] ===== 2. Go to Installed Jres



[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_3.-browse for java7]] ===== 3. Browse for java7 image::images/compatibility-guide-for-java7/compatibility-guide-for-java7-03.png["Browse for java7",width="450",link="images/compatibility-guide-for-java7/compatibility-guide-for-java7-03.png"]

[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_4.-apply changes]] ===== 4. Apply changes image::images/compatibility-guide-for-java7/compatibility-guide-for-java7-04.png["Apply changes",width="450",link="images/compatibility-guide-for-java7/compatibility-guide-for-java7-04.png"]

After following the above instructions, you can import projects or create new ones, and build using java7.

### 3.23.2 Using java8

One can use distribution as is, and there is no extra configuration needed for java8.

### 3.23.3 Using Tomcat8

As mentioned earlier in the guide, distribution comes with *Tomcat8* by default, so no changes are required to run the applications with *tomcat8*.

[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_using-tomcat7 for deploying]] == Using Tomcat7 for deploying

You can download tomcat externally and deploy war in it. For more information, please visit this [link](#).

[[compatibility-guide-for-java7-java8-and-tomcat7-tomcat8\_linux-and windows compatibility]] == Linux and Windows Compatibility

So, the above mentioned steps on *java7 and tomcat7* compatibility, apply to devonfw distributions of Windows OS as well as Linux.

Linux and Windows distribution works by default on **JAVA8** and **TOMCAT8**.

[[devonfwguide\_the-presentation layer: getting started with devon4sencha]] == The Presentation Layer:  
Getting started with Devon4Sencha

## 3.24 Introduction to Devon4sencha



Devon4Sencha is an alternative view layer for web applications developed with Devon Framework. It is based on the proprietary libraries developed by [Sencha](#). As it requires a license for commercial applications it won't be provided as Open Source (The companion OASP project).

These libraries provide support for creating SPA (Single Page Applications) with a very rich set of components for both desktop and mobile. They also provide a clear programming model based on the MVC and MVVM patterns. Sencha also has an excellent documentation for all the APIs in the framework. Visit their [website](#) for more examples on this.

Devon4Sencha builds on top of Sencha libraries to further standardize the way applications are created and to seamlessly integrate with the back-end application. We provide:

- Basic application template
- Basic layout for a tabbed based interface
- Security (CORS)
- Login and session management
- Internationalization
- Better organization of source code (following Sencha MVC pattern)
- Ajax communication simplification

Devon4sencha uses the latest Sencha Version 6. The most important change in the version 6 is that it merges two frameworks: Ext JS (web) and Sencha Touch (mobile) into one single framework. Now, in Ext JS 6, you can maintain a single code. For some of the views, you may need to have a separate view code, but there will be a lot of shared code.

They merged the common code and put them as a core framework, and they brought a concept called toolkit. A toolkit is a package with visual components, such as button, panels, and so on. There are two toolkits: classic and modern. Ext JS visual components are placed in the classic toolkit, and Sencha Touch components are placed in the modern toolkit.

You can simply choose the toolkit that you want to target. If you are writing an application that only targets mobile devices, you can choose modern, and if you are targeting only for desktop, then you can choose the classic toolkit.

### 3.24.1 The universal application

If you want to target both desktop and mobile devices, then in Ext JS 6, you can create a universal application, which will use both the toolkits. Instead of adding the preceding toolkit config mentioned before, you have to add the following builds config section that specifies which build uses which toolkit and theme:

```
"builds": {  
  "classic": {  
    "toolkit": "classic",  
    "theme": "theme-triton"  
  },  
  "modern": {  
    "toolkit": "modern",  
    "theme": "theme-neptune"  
  }  
},
```

The basic idea here is to have two set of toolkits in a single framework in order to target the desktop and mobile devices. When building the application it will create two different bundles for classic and modern.

When accessing from a desktop, the classic bundle will be used. When accessing from a tablet/mobile the modern bundle will be selected.

The sample restaurant application is an example of an **universal** application targeting both toolkits.

## 3.25 Legal considerations

Devon4Sencha is based on Sencha Javascript libraries that are not free and this has to be taken into consideration before starting with a project.

The license model for Sencha is "per developer" and each engagement should take care of having enough licenses for developers working on the view layer.

Client should be informed of this fact and may be provided with licenses also if they retain the IP for the project developed within Capgemini.

## 3.26 Sencha Cmd

To make your Devon4sencha application development easy, you will find, included in the Devon distribution, a tool called Sencha Cmd. It's available for Windows, Mac, and Linux.

[Sencha Cmd](#) is a cross-platform command line tool that provides many automated tasks around the full life-cycle of your applications from generating a new project to deploying an application to production.

It provides a collection of powerful features that work together and in conjunction with the Sencha Ext JS framework like code generation, javascript compiler, workspace management, mobile native packaging, build scripts, theming and so on.

The workspace management feature allows the import of "packages" into your application. That way you can modularize and create packages for sharing functionality between applications. Devon provides one of these packages called `devon-ext.js`.

The most usual way to use Cmd during application development, is just running `sencha app watch` to compile and test the application on a browser. This command must be launched from the application root folder and starts a web server with the client application ready to access from the browser.

Sencha Cmd is not a must for Devon4sencha development application, but using it makes your life easier. So, it's highly recommended to use it.

## 3.27 Workspaces

With [Sencha Cmd Workspaces](#) we can share code and packages between Sencha applications. This is useful because Sencha distribution files occupy quite a bit of disk space.



### Using devcon

You can create a new Sencha workspace using devcon with the `devon sencha workspace` command [learn more here](#)

The easiest way to manually create a Sencha CMD workspace for your application is simply copying the example workspace of devon4sencha (copy the complete directory at `workspaces\examples\devon4sencha` into `workspaces\main\<NAME OF YOUR PROJECT>` and delete the example application's directory called `ExtSample`.

You can also create a workspace from scratch:

- Create the workspace:

```
sencha generate workspace /path/to/workspace
```

- Copy **ExtJS** distribution to `/path/to/workspace/ext`.
- Copy **devon4sencha** package from the example workspace to `/path/to/workspace/packages/devon-ext.js`.

- Copy application template StarterTemplate from the example workspace to /path/to/workspace/StarterTemplate.
- Now, we can create several applications that share the ExtJS distribution and **devon4sencha** package:

```
sencha generate app -ext --starter StarterTemplate MyApp1 /path/to/workspace/MyApp1
sencha generate app -ext --starter StarterTemplate MyApp2 /path/to/workspace/MyApp2
```

### 3.27.1 Sencha packages

With [Sencha Cmd packages](#) we can share and distribute code between Sencha applications. For example, **devon4sencha** is distributed as a Sencha Cmd package that can be shared between applications.

This code will be automatically added to the final application when building the application with Sencha Cmd.

### 3.27.2 Sencha reloading

During application development, we can use the [Sencha Cmd app watch](#) feature to compile and test the application in a browser. This command must be launched from the application root folder.

```
$ cd devon4sencha
$ cd ExtSample
$ sencha app watch
```

Any code changes to the application files will be detected by Sencha Cmd and it will update the application being served. It requires a manual refresh on the browser though to effectively see the changes.

Since this is intended for development and in order to speed up things, only the first "build" defined on app.json will be processed. This means that if you are developing an "universal" app for desktop and mobile you will have several "build" sections defined.

To trigger a concrete "build" you can name it after the `watch` argument

```
$ sencha app watch modern
```

#### Speed up development

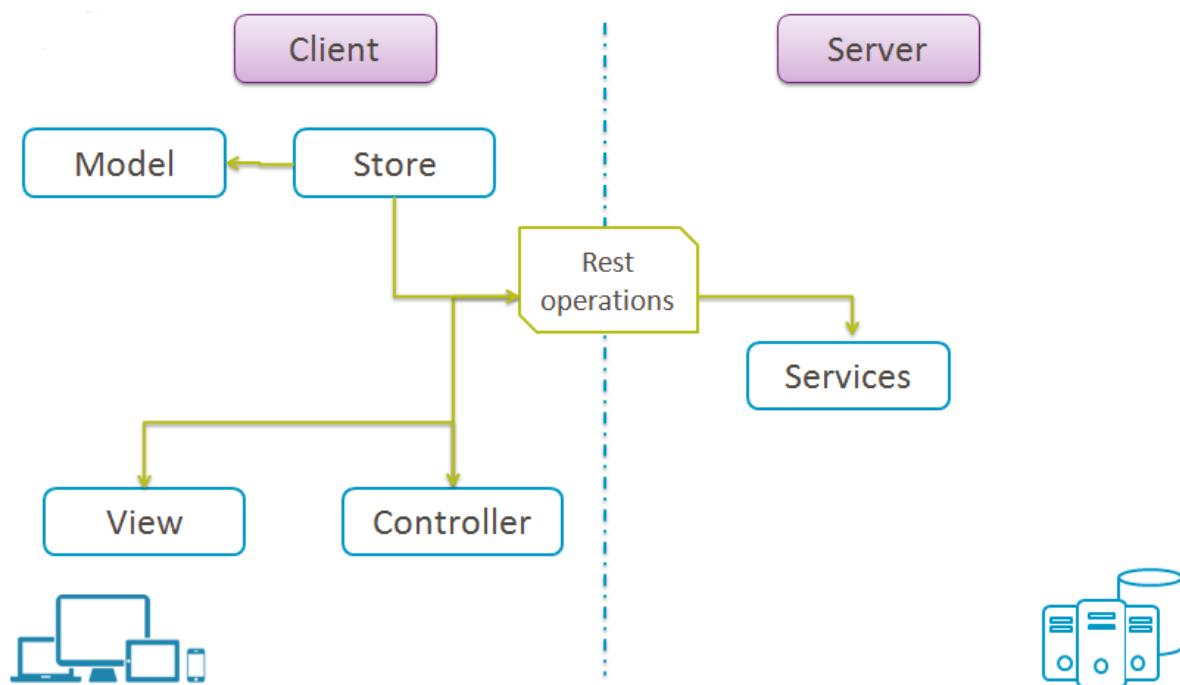
When in development, you can speed things up even more if you are not modifying the sass styles. You can set `skip.sass=1` on the `build.properties` file of your workspace application (i.e: in the restaurant application it would be `devon4sencha/ExtSample/.sencha/app/build.properties`). Don't forget to revert to 0 if you modify styles or change to other build type (classic/modern)

## 3.28 Application architecture

Sencha provides support for both MVC and MVVM application architectures. Please note that this is properly documented on the Sencha docs site. Take this document as an extension of the official documentation to make things more clear.

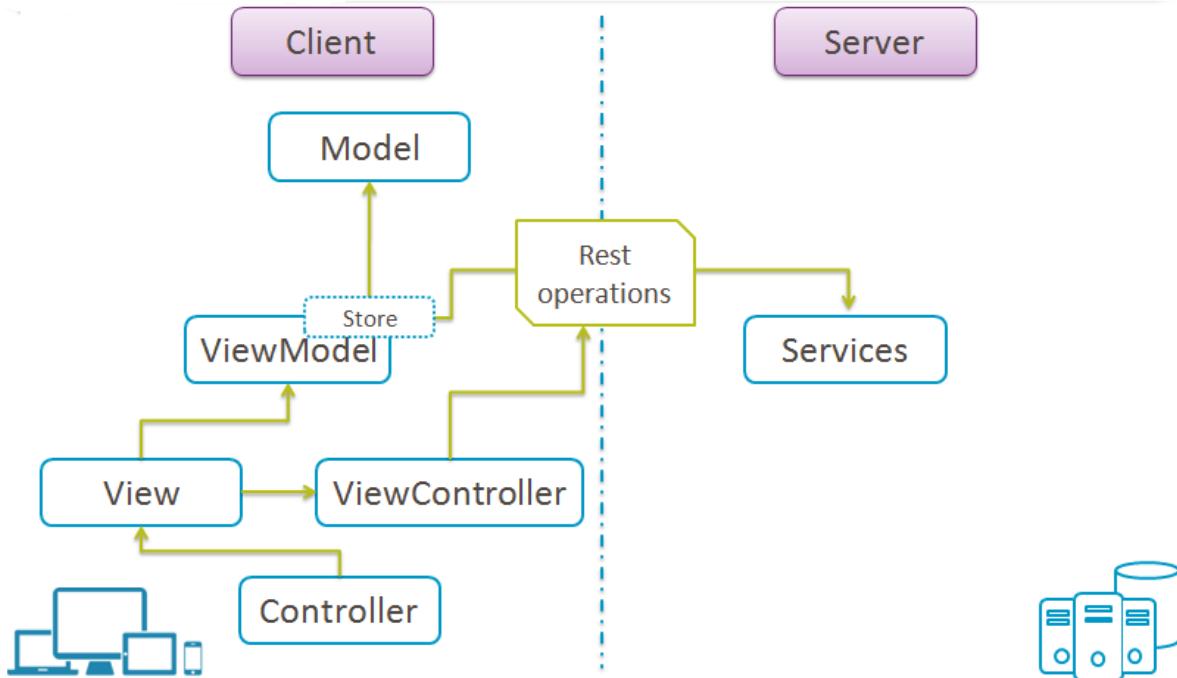
### 3.28.1 MVC (Model, View, Controller):

- User interacts with **Views**, which display data held in **Models**
- Those interactions are monitored by a **Controller**, which then responds to the interactions by updating the **View** and **Model**, as necessary
- **Controllers** will contain the application logic
- **Models** are an interface to data
- The application is easier to test and maintain. Code is more reusable



### 3.28.2 MVVM (Model, View, Controller, ViewModel, ViewController):

- Features an abstraction of a **View** called the **ViewModel**. The ViewModel coordinates the changes between a Model's data and the View's presentation of that data using a technique called "**data binding**"
- The result is that the **Model** and framework perform as much work as possible, minimizing or eliminating application logic that directly manipulates the **View**



## 3.29 Structure of a Devon4sencha application

With these objects in place, Sencha also defines a clean way to structure the application by having each kind of object in separate folders.

Devon4sencha follows the MVVM architecture.

### 3.29.1 Model

The **models** contain the data for the application with some powerful abstractions to retrieve information from the back-end. **Models** have fields defined and associations to other **models** or **stores** which are collections of **models**.

**Models** and **stores** are used by the framework for some controls such as **combo**, **list** and **grid**.

### 3.29.2 Store

A Store is a client side cache of records (instances of a Model class). Stores provide functions for sorting, filtering and querying the records contained within.

### 3.29.3 View

Views are the objects responsible for displaying the data to the user. They contain no logic and have to deal with formatting and internationalization of messages within the components.

Sencha **views** can be created in two ways:

- Programmatic
- Declarative

The most common way to define views is **declarative**, by using Javascript objects that represent the controls with their properties and other child components.

For example:

```
{  
    xtype: 'panel'  
    title: 'parent panel title',  
    items: [{  
        xtype: 'textfield',  
        labelField: 'Label for input',  
    }, {  
        xtype: 'button',  
        text: 'send'  
    }]  
}
```

In this example a component of type `panel` is defined with a `textfield` and a `button` in it. Please note that the behaviour of the button is not specified here.

### 3.29.4 Controller

The controllers are typically the objects responsible for driving the logic of the application, listening to events of the view controls and modifying the underlying model. They are also in charge of rendering new views in response to application state changes.

From version 5 ExtJS this role is usually assumed by the **ViewControllers** and normal **Controllers** are more similar to a **service** pattern on other frameworks. They are typically singletons that are initialized on application launch and are not associated to a concrete view.

In Devon applications, controllers are used for:

- Create **rest endpoint** helper methods on initialization.
- Listening for global events
- Declaring view dependencies
- Instantiating view components
- Specify internationalization bundles

```
Ext.define('Sample.controller.table.TablesController', {  
    extend: 'Ext.app.Controller',  
  
    //Create rest endpoint helper methods on initialization  
    init: function() {  
        Devon.Ajax.define({  
            'tablemanagement.table': {  
                url: 'tablemanagement/v1/table/{id}'  
            },  
            'tablemanagement.search': {  
                url: 'tablemanagement/v1/table/search',  
                pagination : true  
            }  
        });  
    },  
  
    //Listening for global events  
    config: {  
        listen: {  
            global: {  
                eventOpenTableList: 'onMenuOpenTables',  
                eventTableAdd: 'onTableAdd'  
            }  
        }  
    }  
}, {
```

```

//Declaring view dependencies
requires:[
  'Sample.view.table.i18n.Table_en_EN',
  'Sample.view.table.i18n.Table_es_ES',
  'Sample.view.table.TableList',
  'Sample.view.table.TableCrud',
  'Sample.view.table.TableEdit'
], 

//Instantiating view components
onTableEditOrder: function(tableSelected) {
  var id = tableSelected.id;
  var panel = new Sample.view.table.TableEdit({
    title: i18n.tableEdit.title + id,
    viewModel: {
      data: {
        tableViewId: id
      }
    }
  });
  Devon.App.openInContentPanel(panel, {id:id});
}
);

```

### 3.29.5 ViewController

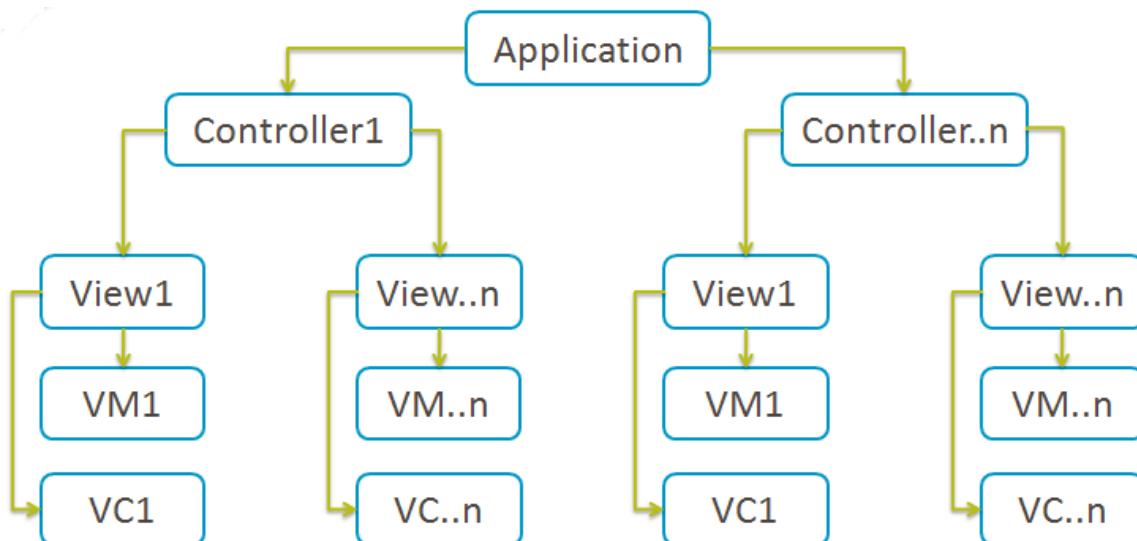
This kind of object is associated to a component view and listens to events of its controls. There is an instance of a **viewController** for each instance of a view so they are created/destroyed as required.

Events of the components of a view are routed to methods of its **ViewController** and the logic of the application is executed.

Usually this means dealing with input control values, calling services of the back-end (maybe through the use of a global **Controller** ) and modifying the **model**.

With this results, the view is modified to reflect those changes.

Having Controllers and ViewControllers could be a bit confusing. Let's have a look at the image below in order to clearly understand how these concepts are organized in a Devon4sencha application:



A Devon4sencha application will have different Controllers. Each of them will control several views related to the same business logic. Each of these Views will have its own ViewController and ViewModel. That ViewController will be in charge of managing specific events for that particular View.

### 3.29.6 ViewModel

The place to store information for a view is the ViewModel. This object contains not only instance data but also **calculated** fields based on other **viewModel**. All this information can be referenced on the view object by means of the **binding** capabilities of the Sencha Framework. One ViewModel object instance is associated to each View instance and they share lifespan.

```
Ext.define('Sample.view.table.TableEditVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.table-edit-model',
    data: {
        orderInfo: null
    }
});

Ext.define("Sample.view.table.TableEdit", {
    extend: "Ext.panel.Panel",
    viewModel: {
        type: "table-edit-model"
    },
    bind : {
        loading : '{!orderInfo}'
    }
});
```

In this sample the loading mask and text will be displayed/hidden according to the value of the viewModel orderInfo data property. This is something that usually requires calling the show/hide methods of the mask object programmatically and by using the MVVM model, this greatly simplified.

## 3.30 Universal applications

When you have to provide a front-end for both desktop and tablet/mobile you can leverage Sencha "universal" application feature. This means that you will have two different outputs from your source code (or more if you have more different profiles to support i.e. TV, POS, smartwatch?)

First thing is to define the "builds" you want to have on the `app.json` file, you will find near the bottom something like this:

```
"builds": {
    "classic": {
        "toolkit": "classic",
        "theme": "theme-crisp",
        "sass": {
            // "save": "classic/sass/save.scss"
        }
    },
    "modern": {
        "toolkit": "modern",
        "theme": "theme-triton",
        "sass": {
            // "save": "modern/sass/save.scss"
        }
    }
}
```

#### Note

You can call `Ext.isModern` or `Ext.isClassic` anytime in your code if you need to know which toolkit are you using at runtime (maybe for showing/hidding elements on a page)

The way "universal" works is by overriding specific files of your application using files specific for those "builds". This is done by having one folder per "build" type on the application root folder replicating the application usual structure. Let's see an example.

In a "normal" application you will have this structure:

```
app
+ controller
+ model
+ store
+ view
  + main
    Main.js
    MainVC.js
    MainVM.js
```

Now, if we have two builds called "classic" and "modern" and want to have specific views defined for Main component we will have this structure:

```
app
+ controller
+ model
+ store
+ view
  + main
    MainVC.js
    MainVM.js
classic
+ src
  + view
    + main
      Main.js
modern
+ src
  + view
    + main
      Main.js
```

## 3.31 Devon4sencha Sample application

Devon comes with a sample java application simulating the backend for a restaurant application. Devon4sencha's sample application completes the restaurant application with a user-friendly UI.

### 3.31.1 Running the sample restaurant application

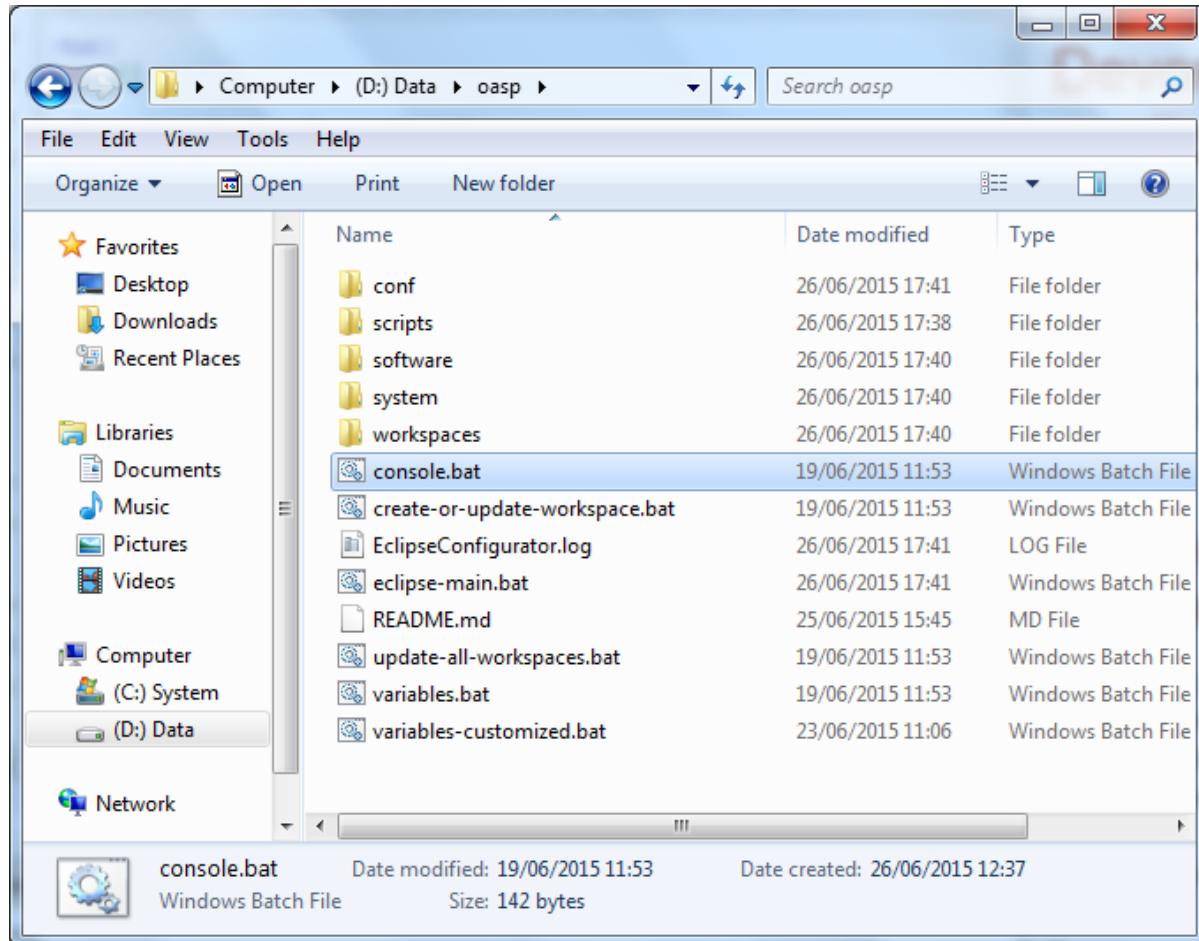


#### Using devcon

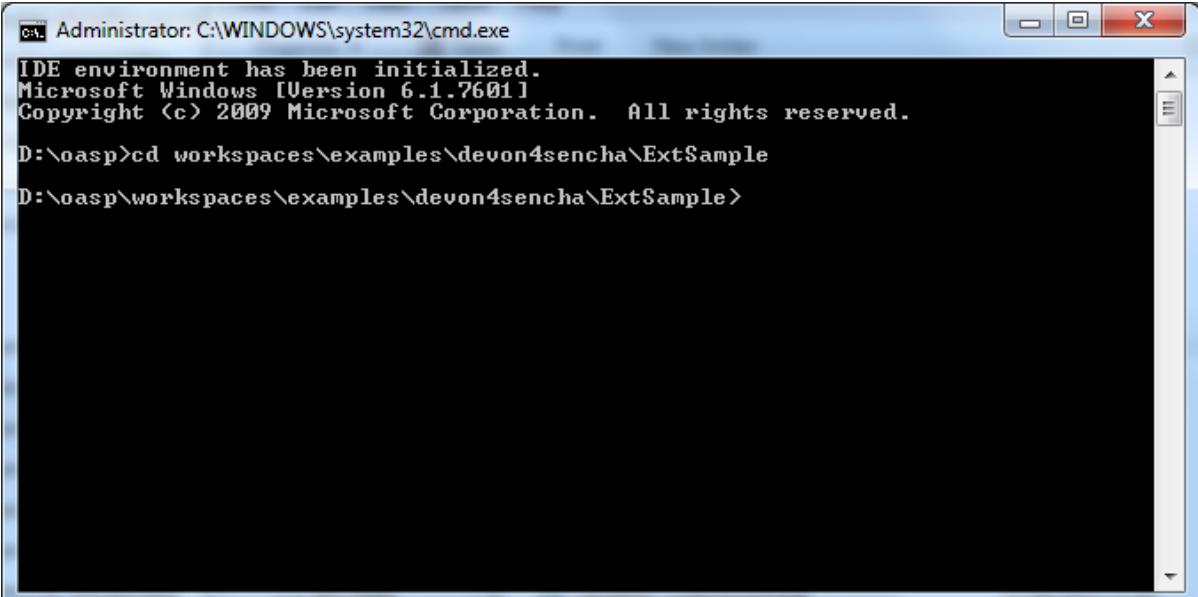
You can automate the following steps using devcon with the `devon sencha run` command [learn more here](#)

The following steps assume that you already have setup your Devon environment, and that you have the `devonfw-sample-server` running on a server on port 8081. Refer to the Devonfw Server documentation for instructions on how to setup the Devon environment or the devon sample application.

1. Open a Devon command prompt by executing the batch file `console.bat`.



2. Enter the `workspaces\examples\devon4sencha\ExtSample` directory



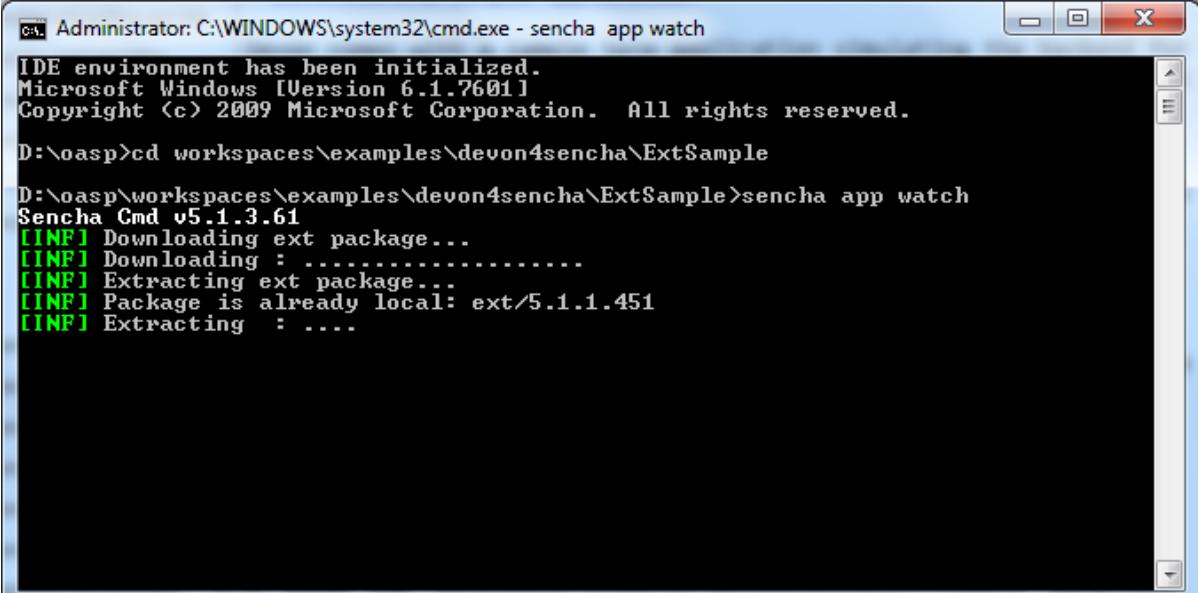
```
Administrator: C:\WINDOWS\system32\cmd.exe
IDE environment has been initialized.
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

D:\oasp>cd workspaces\examples\devon4sencha\ExtSample
D:\oasp\workspaces\examples\devon4sencha\ExtSample>
```

3. Execute the command `sencha app watch`. It might take some time the first time you run this command.

This command will compile the devon4sencha sample application and start a webserver to serve it and will automatically recompile the application if it detects any changes to the application's files.

On first run, you probably will see something similar to the image below. We do not distribute ExtJS itself with our sample application. Instead, the ExtJS SDK is downloaded on first use if not already available.



```
Administrator: C:\WINDOWS\system32\cmd.exe - sencha app watch
IDE environment has been initialized.
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

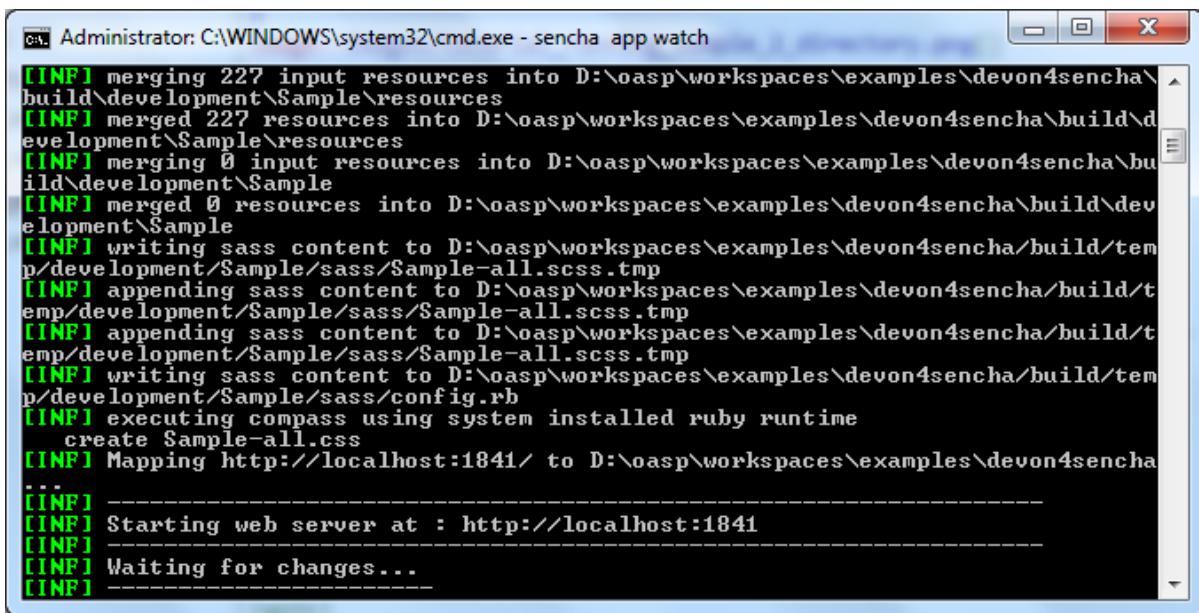
D:\oasp>cd workspaces\examples\devon4sencha\ExtSample
D:\oasp\workspaces\examples\devon4sencha\ExtSample>sencha app watch
Sencha Cmd v5.1.3.61
[INF] Downloading ext package...
[INF] Downloading : .....
[INF] Extracting ext package...
[INF] Package is already local: ext/5.1.1.451
[INF] Extracting : .....
```

If you have a valid ExtJS license, you can simply copy the ExtJS 6 SDK into `workspaces\examples\devon4sencha\ext` and it will automatically get picked up.

### Note

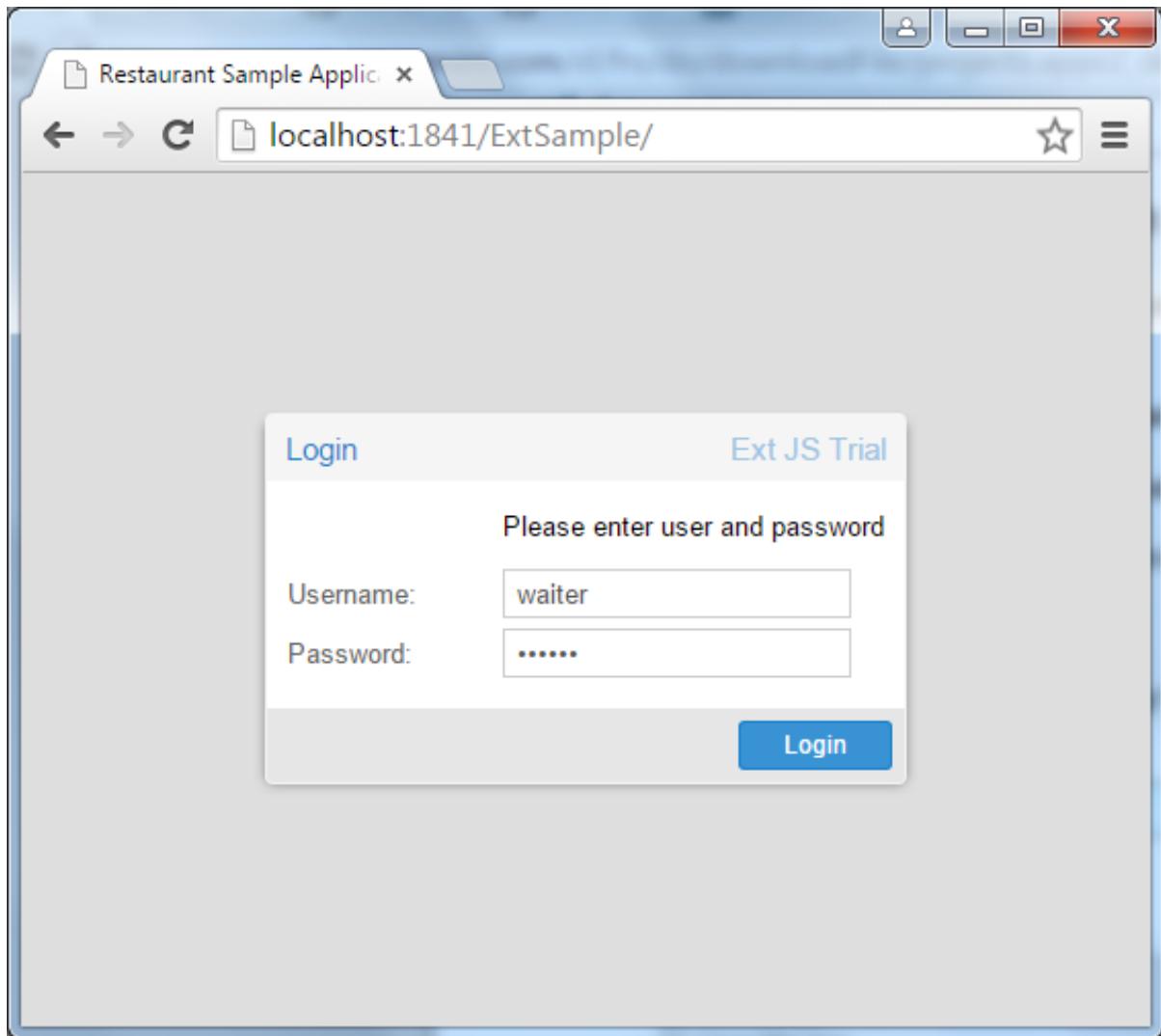
If you see an error while executing this command, first try to delete the local sencha repository located at <DEVON ENVIRONMENT>\software\Sencha\Cmd\default\repo. If there was a problem downloading the ExtJS SDK, the local repository sometimes gets stuck in a bad state, and the easiest solution is to just delete it, causing everything to be re-downloaded.

4. As soon as you see Waiting for changes..., the application has been compiled and is ready.



```
[INFO] merging 227 input resources into D:\oasp\workspaces\examples\devon4sencha\build\development\Sample\resources
[INFO] merged 227 resources into D:\oasp\workspaces\examples\devon4sencha\build\development\Sample\resources
[INFO] merging 0 input resources into D:\oasp\workspaces\examples\devon4sencha\build\development\Sample
[INFO] merged 0 resources into D:\oasp\workspaces\examples\devon4sencha\build\development\Sample
[INFO] writing sass content to D:\oasp\workspaces\examples\devon4sencha\build\temp\development\Sample\sass\Sample-all.scss.tmp
[INFO] appending sass content to D:\oasp\workspaces\examples\devon4sencha\build\temp\development\Sample\sass\Sample-all.scss.tmp
[INFO] appending sass content to D:\oasp\workspaces\examples\devon4sencha\build\temp\development\Sample\sass\Sample-all.scss.tmp
[INFO] writing sass content to D:\oasp\workspaces\examples\devon4sencha\build\temp\development\Sample\sass\config.rb
[INFO] executing compass using system installed ruby runtime
  create Sample-all.css
[INFO] Mapping http://localhost:1841/ to D:\oasp\workspaces\examples\devon4sencha...
[INFO] -----
[INFO] Starting web server at : http://localhost:1841
[INFO] -----
[INFO] Waiting for changes...
```

5. Open <http://localhost:1841/ExtSample/> in a browser. Use waiter as both user and password.



## 3.32 Creating a new application

For creating a new application, we start by creating a Sencha CMD workspace to host our project files.



### Using devcon

You can automate these processes using devcon. With the `devon sencha workspace` command you can generate new Sencha workspaces [learn more here](#).

With the `devon sencha create` command you can create new Sencha applications [learn more here](#).

If you are going to do it manually copy the example workspace at `workspaces\examples\devon4sencha` and then remove the sample application by deleting the directory `ExtSample`.

Then we are set to automate the application creation with the help of Sencha Cmd. We provide an application template, which already integrates Devon4sencha in your new application. Use it by issuing the following command inside of your Sencha workspace:

```
sencha generate app -ext --starter StarterTemplate <NAME OF YOUR APP> <DIRECTORY FOR YOUR APP>
```

This will generate the structure of the project with the basic setup of a devon4sencha application.

In our case, for this guide purpose, the name of the app will be **demo** and the directory for the app will be **demo** as well. This will generate the structure of the project with the basic setup of a devon4sencha application.

The application has been created using the template `StarterTemplate`. This template is located in the workspace:

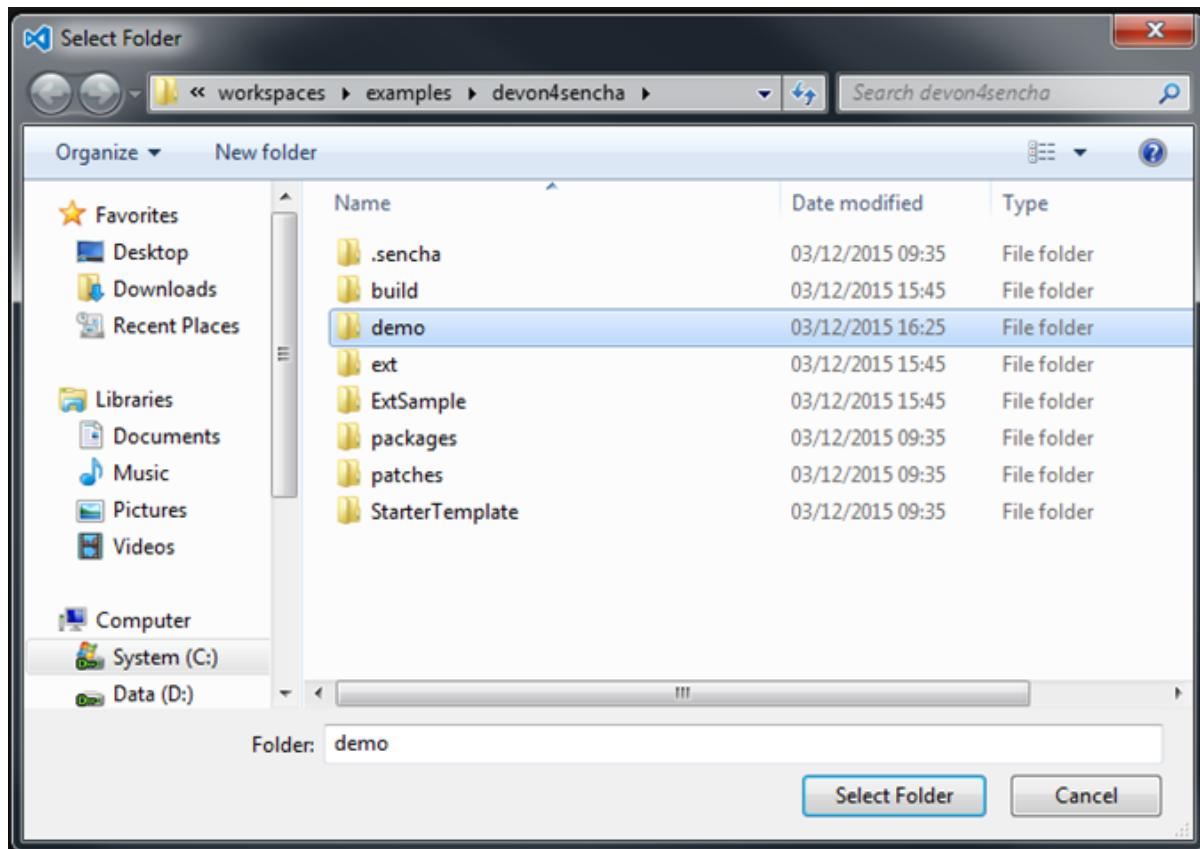
```
<devon_distribution>\workspaces\examples\devon4sencha\StarterTemplate
```

Once the application has been created we are going to open our development IDE. Although you can use any simple text editor to write the Ext JS code, using the IDEs definitely makes it a bit easier. These are some of the IDEs recommended:

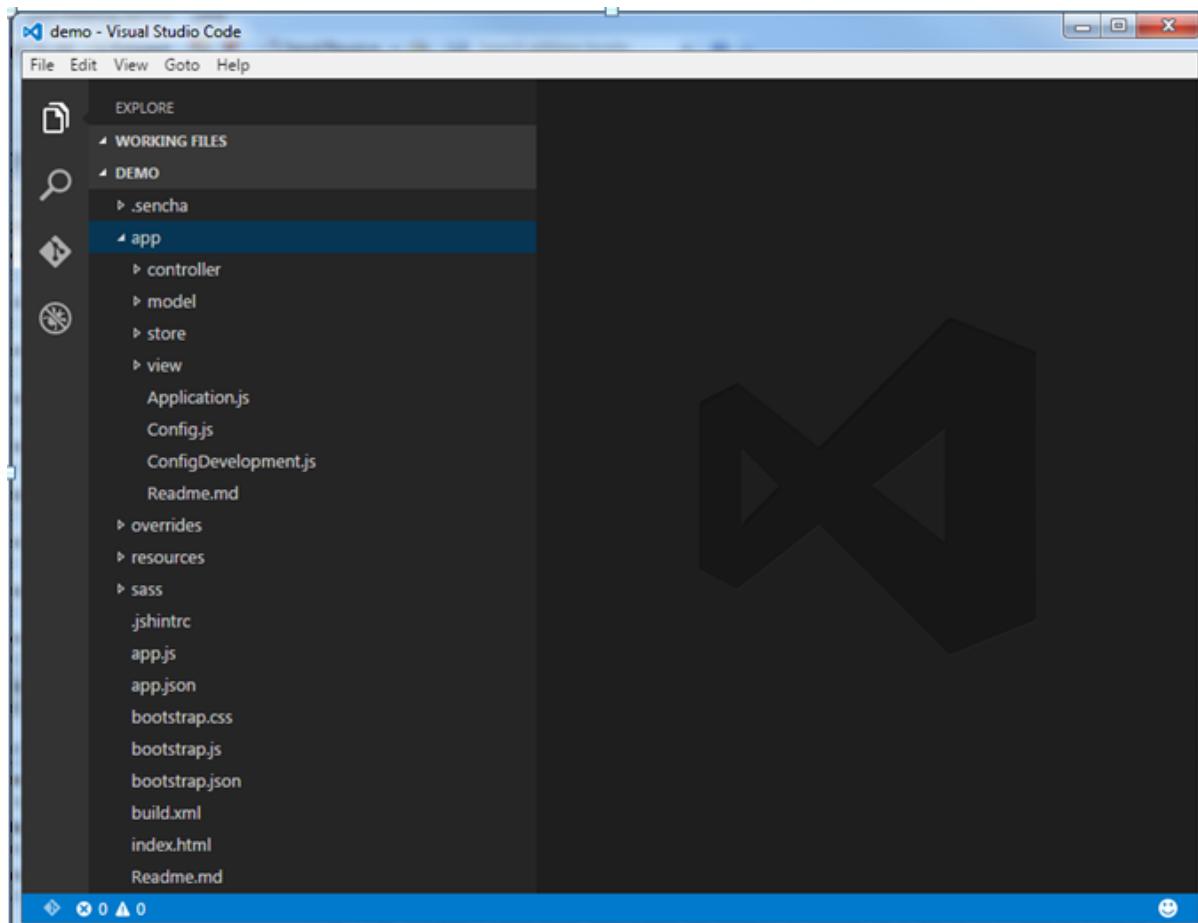
- Visual Studio Code
- Atom
- Sublime
- Notepad++
- WebStorm

In this guide we are going to use Visual Studio Code. You can download it from their [website](#)

With the Visual Studio Code we have to open the folder of our application. Then we click File#Open Folder and we check 'demo' as is the folder of the application.



We should be seeing a file structure like this one:



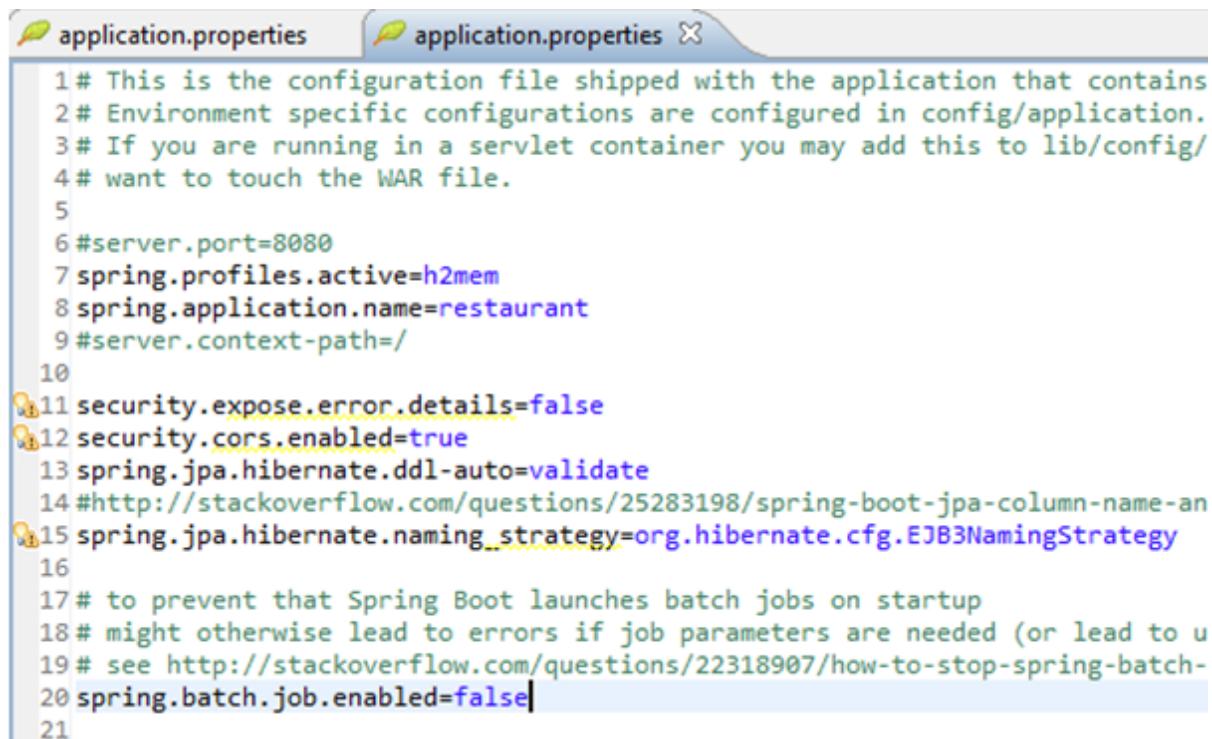
You can learn more about the Devon4sencha app structure [here](#).

For this guide, we are going to use the same backend as in the sample application so we have to change the path of the server in the ConfigDevelopment.js file:

```
window.Config.server = 'http://localhost:8081/devonfw-sample-server/services/rest/';  
window.Config.CORSEnabled = true;
```

For your particular application, please change here the url to point your server side. In case you are trying to point a new Devonfw Server application, you should change the CORS configuration in the server-side:

- Open the file <project\_name>-core//src/main/resources/application.properties and change the property 'security.cors.enabled'. By default is false and you should change it to true.
- Stop SpringBoot and start it again.



```
1# This is the configuration file shipped with the application that contains
2# Environment specific configurations are configured in config/application.
3# If you are running in a servlet container you may add this to lib/config/
4# want to touch the WAR file.
5
6#server.port=8080
7spring.profiles.active=h2mem
8spring.application.name=restaurant
9#server.context-path=/
10
11security.expose.error.details=false
12security.cors.enabled=true
13spring.jpa.hibernate.ddl-auto=validate
14#http://stackoverflow.com/questions/25283198/spring-boot-jpa-column-name-an
15spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.EJB3NamingStrategy
16
17# to prevent that Spring Boot launches batch jobs on startup
18# might otherwise lead to errors if job parameters are needed (or lead to u
19# see http://stackoverflow.com/questions/22318907/how-to-stop-spring-batch-
20spring.batch.job.enabled=false
21
```

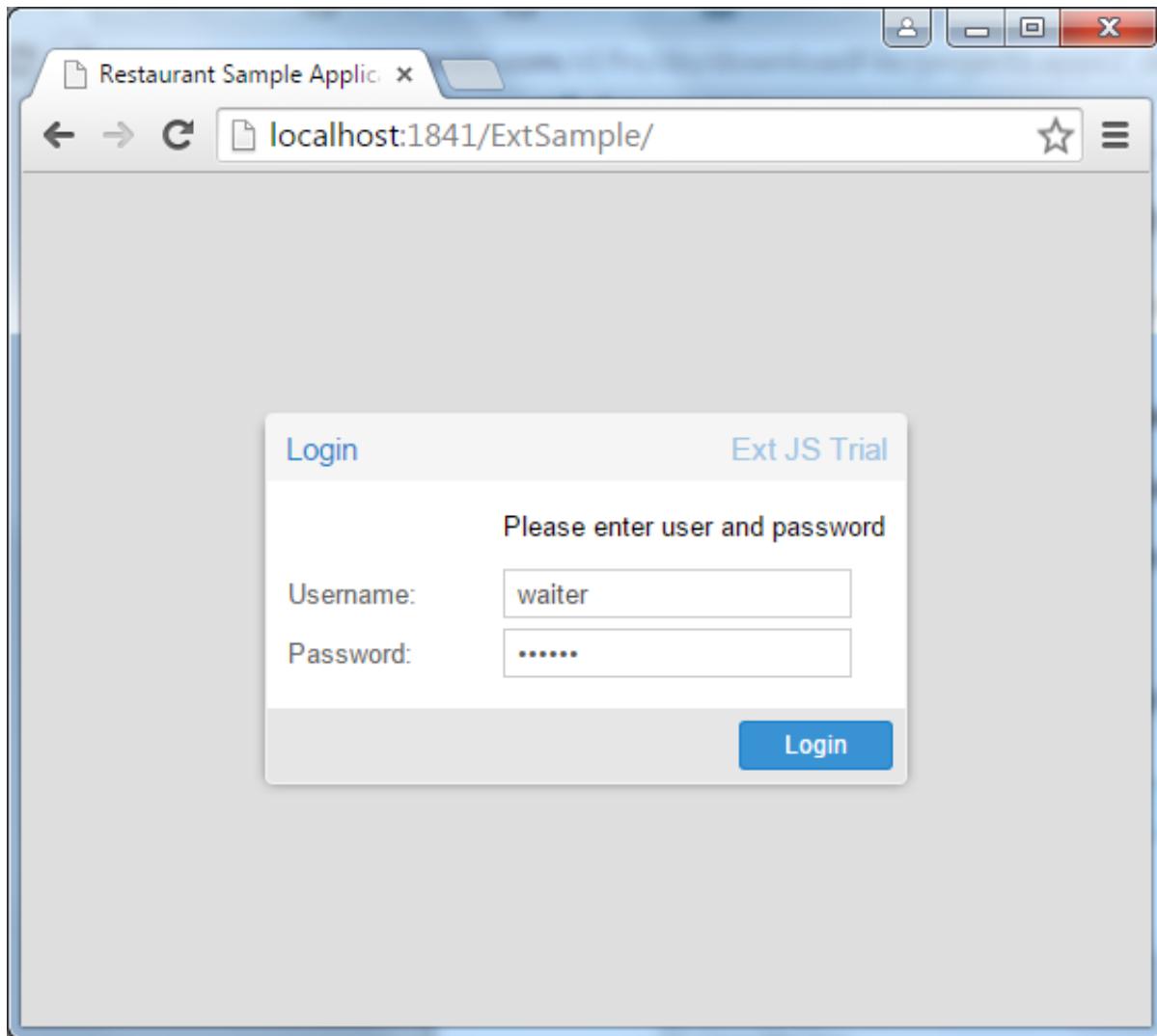
We can already run the application from the command line by entering our app's directory and issuing:

```
sencha app watch
```

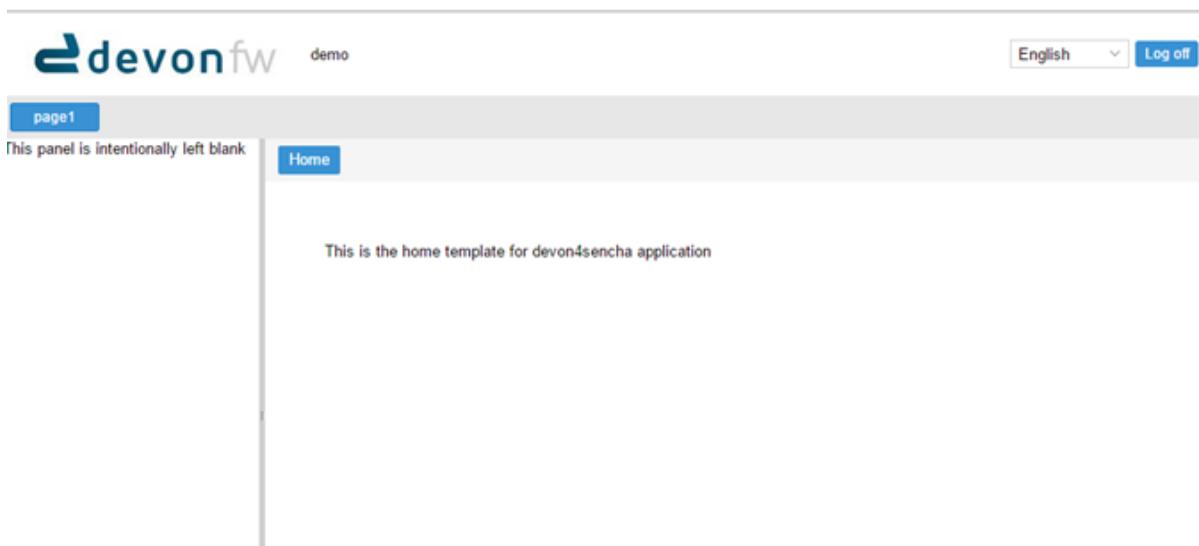
### Using devcon

Remember that you can automate this process using devcon with the devon sencha run command [learn more here](#)

The result should be the same login page as in the sample application:



If we click the Login button we will enter into the demo application:



This is what Devon4Sencha provides by default as a Template for the client application.

## 3.33 Project Layout

The layout of folder/files recommended to work with a Sencha project in a MVVM fashion is as follows:

```
app/controller
  /model
  /store
  /view
  /Application.js
app.json
Config.js

packages
```

This structure is understood by Sencha Cmd in order to provide facilities to work with the code. As you can observe there is no html as it will be generated by Cmd when building the application.

- **Folder controller:** This folder contains the main controllers of the application.
- **Folder model:** This folder contains the models of the application.
- **Folder view:** This folder contains the views, viewControllers, viewModels of the application.
- **File Application.js:** The main application class based on Devon.App(Ext.app.Application) class.
- **File app.json:** Sencha Cmd configuration file.
- **File Config.js:** Config javascript configuration properties. It contains all configuration properties needed by the application like internationalization, REST URL endpoints, etc. This file must be loaded before the rest of the application (app.json takes care of this). For example:

```
window.Config={
  defaultLocale:'en_EN',
  supportedLocales:['en_EN','es_ES'],
  server : '/devonfw-sample-server/services/rest/',
  CORSEnabled: false
};
```

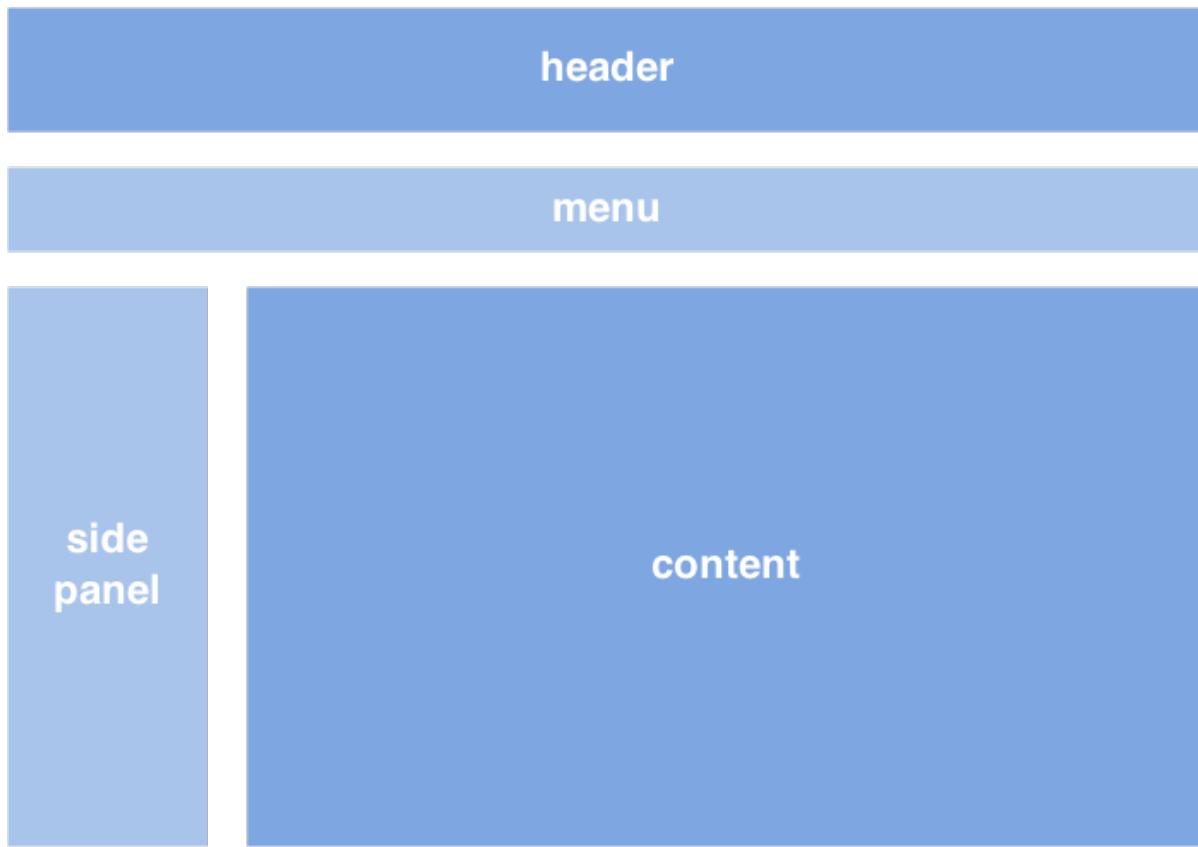
- **Folder packages:** In this folder resides the devon4sencha package that meets the application requirements. In \packages\local\devon-extjs you can find what Devon4Sencha builds on top of Sencha libraries.

## 3.34 Main Template

Devon4Sencha provides the basis for creating SPA (Single Page Applications) both for desktop and mobile clients.

For each device there is an appropriate layout that is more suitable for big or small screens. Devon proposes a template for starting an application that can be customized or completely replaced if needed.

For desktop, the main layout of the application is as follows:



Here we find:

- Header: for displaying the logo, application name, user related info, etc...
- Menu: main navigation controls for the application, with sub-menus
- Side panel: area dedicated to secondary functionality of the application
- Content: main area where the "pages" of the application are loaded

This pattern allows for very productive applications because the content area makes use of a Tab component that can display multiple instances of different entities at once, so the user can rapidly cycle between them, or load several in parallel.

Sencha layout system works in a similar way to traditional Java "swing" applications composing visual controls (called components) into containers that can be arranged into other containers.

The main view for the application is called **viewport** and Devon provides one by default **Devon.view.main.Viewport**

Inside this **viewport** all the areas of the template are included if defined by the application based on the **alias** of the containers. This alias allows to reference the component within other components or containers. The **alias** used on the viewport are:

- main-viewport
- main-header

- main-menu
- main-slidepanel
- main-content

If the application doesn't declare a container with such an **alias** then it won't be shown on the application.

If more customization is needed then it is better to not start with `Devon.view.main.Viewport` and create your own `viewport` object.

Another concern for the **viewport** is to contain references to **global** data that can be addressed by visual components of the application, such as information about the logged user. This is achieved by storing this data into the **ViewModel** of the **viewport**. As the **ViewModel** is inherited by contained components, storing information at the root of the view hierarchy makes this available everywhere.

This can be useful for example for [controlling visibility of controls based on user roles](#)

### 3.34.1 Displaying pages

The main template for Devon applications is based on a tabbed layout. This is very convenient and makes for very productive applications since several entities can be opened at the same time and the user can switch easily between them.

Displaying pages on this template is only a matter to add children panels to this `tabbar` which can be addressed by its alias `main-content` or by using the Devon method `Devon.App.openInContentPanel` (see `jsdoc` for more information on the usage of this method)

#### Closing pages with forms without losing input data

Devon framework offers the `Devon.plugin.PreventDataLoss` plugin valid only for `Ext.form.Panel` objects. This plugin alerts the user about losing data when the form has been edited on screen and hasn't been saved, before closing a page or window. This plugin can be used as shown below:

```
Ext.define('Sample.some.View', {
    extends: 'Ext.form.Panel',

    closable: true,
    bind: {
        values: '{myValues}'
    },
    plugins: ['preventdataloss'],
    items: [
        {
            xtype: 'textfield',
            reference: 'id',
            name: 'id',
            bind: {value: '{myValues.id}'}
        }
    ];
});
```

Note that the form, or a panel or tab that contains the form, must be **closable**.

In a form panel, a record or an object with data can be used to bind to the view properties `record` or `values` (as shown in the sample).

By specifying the property `mainPanel` (String: panel `xtype`), the plugin will search for a parent component with that `xtype`, and alerts the user before closing the referenced panel although it doesn't need to be closable.

```
Ext.define("Sample.view.table.TableCrud", {
    extend: "Ext.panel.Panel",
    xtype:'tablecrud',

    items:[{
        xtype:'form',
        bind:{
            values:{table}
        },
        plugins:[{
            ptype:'preventdataloss',
            mainPanel:'tablecrud'           // <-- component to watch for modifications
        }],
        items:[]
    }]
});
```

## 3.35 Code conventions

We suggest the following conventions to work with devon4sencha:

### 3.35.1 Controllers, ViewModels and ViewControllers

The filename of these classes must be the same as the principal view and must end with `VM` for ViewModels and `VC` for ViewControllers:

```
MyViewV.js
MyViewVM.js
MyViewVC.js
```

The files will reside on the same folder as the view.

For controllers, the convention is to end the file name with "Controller" and the files should reside on a parallel folder structure similar to the view but on the `controller` app sub-folder.

### 3.35.2 Internationalization

Internationalization files must reside in a `i18n` folder besides the associated view. The filenames must start with the associated view name end with the corresponding locale:

```
MyViewV.js
MyViewVM.js
MyViewVC.js
i18n/MyView_en_EN.js
i18n/MyView_es_ES.js
```

### 3.35.3 Event Names and listeners

For events and listeners definitions, we suggest that event names start with `event` and listener function names start with `on`. The rest of both names will use [camelcase format](#):

```
config: {
  listen: {
    global: {
      eventOpenTableList: 'onMenuOpenTables'
    }
  }
},
onMenuOpenTables : function(){
  //....//
}
```

### 3.35.4 Code style

In general we follow the code style used by Sencha in their frameworks, for example:

- 4 spaces for indentation (NO tabs)
- 1 class per file

We provide configuration files for JSCS (Javascript Checkstyle) and jshint (Javascript linter) in the devon4sencha distribution.

## 3.36 Sencha Client and Sencha Architect project generation

The generation can create a full EXT JS 6 client using the StarterTemplates of the Devon4Sencha package and a Sencha Architect project as well. For more details about the project layout and the template used, please refer to Devon4Sencha documentation [here](#).

### 3.36.1 Getting ready

## Sencha Workspace and App

For a new EXT JS 6 project:

## Step 1: Devon4Sencha workspace

Copy from workspaces\examples of your DevonFW distribution the devon4sencha folder into the core folder of your OASP4J project.

In windows you can use a "junction" to create a symbolic link to the devon4sencha folder instead of copying it. To leverage this NTFS feature you need to use a external tool (<https://technet.microsoft.com/en-us/sysinternals/bb896768.aspx>) since Windows doesn't provide an utility to create them

Using the restaurant example, the project structure will be like this:

```
oasp4j/
    samples/
        core/
            src/main/java/
                devon4sencha/
```

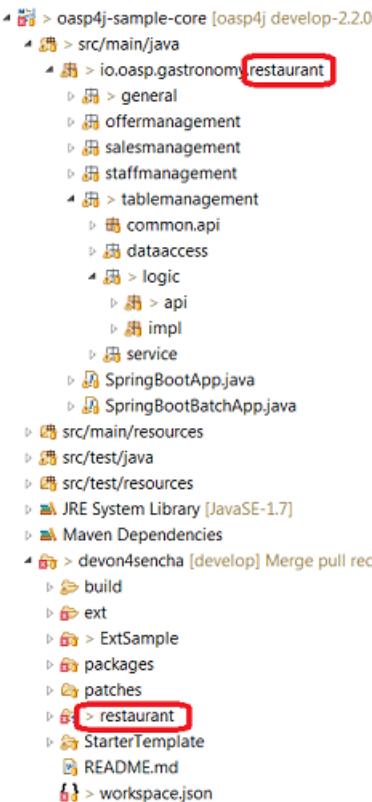
## Step 2: Create the app

Open a console and navigate to the decon4sencha directory from the core folder and write the following command, being APP\_NAME the same name as the last part of the basePackage, the name of the package where the `SpringBootApp` class is:

```
sencha generate app -ext {APP NAME} {APP NAME}
```

For example:

```
sencha generate app -ext restaurant restaurant
```



Refresh your OASP4J project (F5).

### Step 3: Config.js and ConfigDevelopment.js

Edit this two files to configure the endpoint where the ajax calls have to be made. ConfigDevelopment overrides the values in Config.js but it is skipped when doing production builds.

During development you will be deploying client and server code independently, this is more productive since you avoid building the whole package only for changes on the JavaScript files. What this is that JavaScript code has to make ajax request to a different domain to where the code is being called.

To enable this and avoid getting cross domain browser errors you have to enable CORS on the server side.

### CobiGen Templates

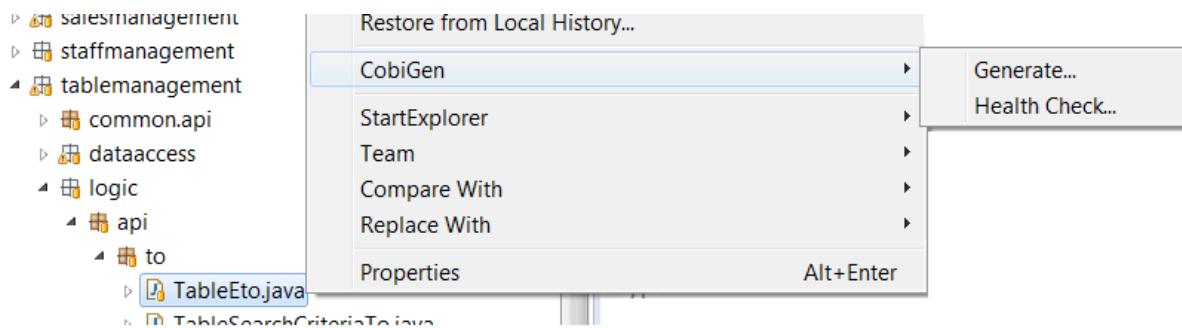
Before begin the generation, import the CobiGen\_templates project included at yor DevonFW distribution into your Eclipse work space

The project has the templates for the Sencha Client and Sencha Architect project generation.

## 3.36.2 Generating

### Input

The input file for the generation must be the **Eto** file because is the object transferred between the server and the client.



## Wizard

If the templates are imported at the Eclipse workspace, and the input file is an **Eto**, the wizard will open, this increments can be chosen:

- Sencha Client App
- Sencha Architect Project

For the first time generation, the APPNAME\_architect\_project folder will be created at the root of the origin project with all the folders and files generated. After that, all the next generations will be merged at thus folders.

### 3.36.3 Deploying

- To deploy the Sencha Client App:

1. Just run over the APPNAME folder from the console the following command:

```
sencha app watch
```

- To deploy the Sencha Architect Project

1. just double click over the .xds file or opening it with the Sencha Architect menu.

## 3.37 Creating a new page

Now we will create a sample page with an associated ViewModel and ViewController which will be shown on the menu of the application. The main objective here is to show you quickly with a simple example all the steps we should follow in order to create a new view and the functionality associated.

Please follow these steps:

### 3.37.1 Step 1: ViewModel

Create a new ViewModel in the file `app/view/NewPageVM.js`:

```
Ext.define('demo.view.NewPageVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.new-page',
    data: {
        htmlContent: 'My page content'
    }
});
```

We will make use of the data of this ViewModel within the view object.

### 3.37.2 Step 2: ViewController

Create a new ViewController in the file `app/view/NewPageVC.js`:

```
Ext.define('demo.view.NewPageVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.new-page',
    onTestButtonClick: function(){
        alert('Click on test button'
    }
});
```

### 3.37.3 Step 3: View

Create the new view object in the file `app/view/NewPage.js`. Here we request and assign the ViewModel and the ViewController previously created:

```
Ext.define('demo.view.NewPage', {
    extend: 'Ext.Panel',
    alias: 'widget.new-page',
    requires: [
        'demo.view.NewPageVM',
        'demo.view.NewPageVC'
    ],
    controller: 'new-page',
    viewModel: 'new-page',
    title: 'New page',
    bind: {
        html: '{htmlContent}'
    },
    buttons: [
        {
            text: 'Test button',
            handler: 'onTestButtonClick'
        }
    ]
});
```

Please note how we make use of the `bind` property on the view to access the data of the viewModel.

Also for the button on the page where we reference the `onTestButtonClick` handler defined on the `ViewController`, there is no need to specify that the handler is for the `click` event because it's the default for the `button` component.

### 3.37.4 Step 4: MenuItem

In order to have the view appear in the menu of the application we should add an item to the view object. If the recommended layout for Devon application is used we have to edit the `app/view/Menu.js` file:

```
Ext.define('demo.view.main.Menu', {
    extend: 'Ext.Panel',
    alias: 'widget.main-menu',
    requires: [
        'Ext.toolbar.Toolbar',
        //by default use the Devon VC for this menu
        'Devon.view.main.MenuVC'
    ],
    controller: 'main-menu',
    cls: 'main-menu',
    buttonAlign: 'left',
    buttons: [
        {
            text: 'page1', //i18n
            eventName: 'eventOpenPage1'
        },
        {
            text: 'My new page',
            eventName: 'eventOpenMyNewPage'
        }
    ]
});
```

Here, the `eventName` property is not part of Sencha but **Devon4sencha**. It will be used by the `Devon.view.MenuVC` instance (if you use the recommended layout) to fire an event when the menu option is selected. This event will be caught by the following controller.

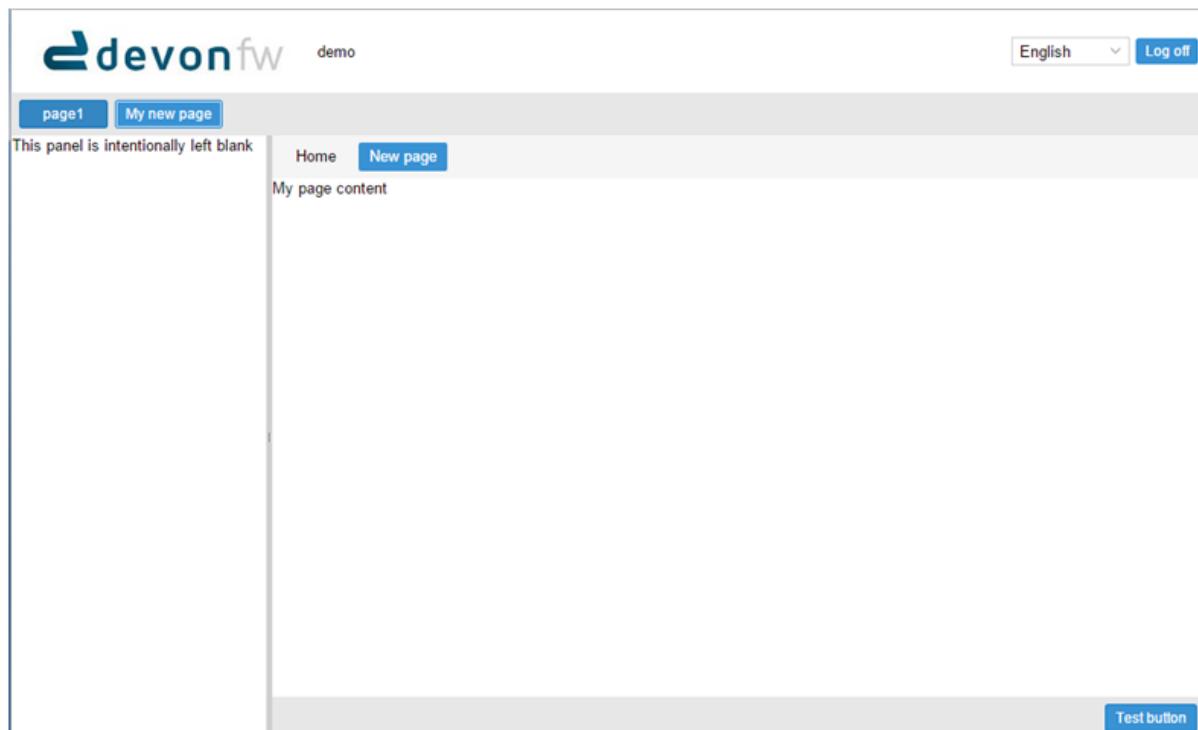
### 3.37.5 Step 5: Controller

Edit the `app/controller/MainController.js` file to add the `eventOpenMyNewPage` event listener function. This instantiates and shows the new page by making use of the `Devon.App.openInContentPanel` method. We also have to add the new view dependency:

```
Ext.define('demo.controller.main.MainController', {
    extend: 'Ext.app.Controller',
    requires: [
        'demo.view.main.i18n.Main_en_EN',
        'demo.view.main.LeftSidePanel',
        'demo.view.main.Content',
        'demo.view.main.Home',
        'demo.view.NewPage'
    ],
    config: {
        listen: {
            global: {
                eventOpenMyNewPage: 'onOpenMyNewPage'
            }
        }
    },
    onOpenMyNewPage: function() {
        var view = Ext.widget("new-page");
```

```
        Devon.App.openInContentPanel(view);
    },
    onClickButton: function () {
        Ext.Msg.confirm('Confirm', 'Are you sure?', 'onConfirm', this);
    },
    onConfirm: function (choice) {
        if (choice === 'yes') {
            //
        }
    });
});
```

Navigate to the application in the browser and check the result:



## 3.38 Create a CRUD page

In this section a complete CRUD (Create, Retrieve, Update, Delete) entity example is developed with Devon4sencha step by step. We are going to do a CRUD for a `table` entity.

### 3.38.1 CRUD Packaging

As an overview, these are the folders and files structure we need for the example.

```
app/
+- controller/
|   +- table/
|   |   +- TableController.js
+--- model
|   +- table
|   |   +- TableM.js
+- store
|   +- table
|   |   +- TableS.js
+- view
|   +- table
|   |   +- TableEditV.js
|   |   +- TableEditVC.js
|   |   +- TableEditVM.js
|   |   +- TableListV.js
|   |   +- TableListVC.js
|   |   +- TableListVM.js
|   +- i18n
|       +- Table_en_EN.js
```

### 3.38.2 Step 1: Add a CRUD page to the application menu

The first step is to add a new item to our application menu.

The code of the `Menu.js` file in the path `app/view/main/` should be:

```
Ext.define('demo.view.main.Menu', {
    extend: 'Ext.Panel',
    alias: 'widget.main-menu',
    requires: [
        'Ext.toolbar.Toolbar',
        //by default use the Devon VC for this menu
        'Devon.view.main.MenuVC'
    ],
    items: [
        {
            xtype: 'toolbar',
            items: [
                {
                    text: 'Tables', // i18n.main.menu.tables,
                    eventName: 'eventOpenTableList'
                }
            ]
        },
        {
            controller: 'main-menu',
            cls: 'main-menu',
            buttonAlign: 'left',
            buttons: [
                {
                    text: 'page1', //i18n
                    eventName: 'eventOpenPage1'
                },
                {
                    text: 'My new page',
                    eventName: 'eventOpenMyNewPage'
                }
            ]
        }
    ]
});
```

```
    ]  
});
```

We are adding a new item in the menu, with an internationalized text (later we will talk about internationalization). When this menu option will be pressed, an event `eventOpenTableList` will be fired so we need an object to listen for this event: The controller.

### 3.38.3 Step 2: Controller, a view factory and REST endpoints definition

Now, we are going to create a global controller to manage all global logic about the different table's views. Basically, this controller will be a factory to create instances of table's views and it will have the REST endpoints definition.

A Controller is also needed besides a ViewController because the later is tightly coupled to a view and only exists when the view exists... so there has to be someplace responsible to instantiate the view.

Controllers in Sencha are global and we try to decouple between controllers by using events and not referencing each other directly. This helps us to modularize applications.

We create a `TableController.js` file in the path `app/controller/table` with the follow content:

```
Ext.define('demo.controller.table.TableController', {  
    extend: 'Ext.app.Controller',  
    /*view requires*/  
    requires:[  
        'demo.view.table.i18n.Table_en_EN',  
        'demo.view.table.TableListV'  
    ],  
    /*Global events listeners definition*/  
    config: {  
        listen: {  
            global: {  
                eventOpenTableList: 'onMenuOpenTables'  
            }  
        }  
    },  
    /*Rest end points definition*/  
    init: function() {  
        Devon.Ajax.define({  
            'tablemanagement.table': {  
                url: 'tablemanagement/v1/table/{id}'  
            }  
        });  
        /*Create a view when a global event is fired*/  
        onMenuOpenTables: function() {  
            var tables = new demo.view.table.TableListV();  
            Devon.App.openInContentPanel(tables);  
        }  
    }  
});
```

As we can see, `TableController` is listening for the `eventOpenTableList` event and when it will be fired, the controller will call the function `onMenuOpenTables` to create an instance of the view `TableListV` (we will define this view later). In addition, we have defined functions to create instances of `TableEditV`. Also, we have defined one REST endpoint to do CRUD operations.

We have to include in `Application.js` the new Controller we have just created:

```
Ext.define('demo.Application', {  
    extend: 'Devon.App',  
  
    controllers: [  
        'demo.controller.main.MainController',
```

```
'demo.controller.page1.Page1Controller',
'demo.controller.table.TableController'
],
name: 'demo',
stores: [
// TODO: add global / shared stores here
],
launch: function () {
// TODO - Launch the application
}
});
```

### 3.38.4 Step 3: Create a model

Before we create the views, we are going to define the table model. This model contains the definition of every field in a table object. We create the file `TableM.js` in the path `app/model/table/`.

```
Ext.define('demo.model.table.TableM', {
extend: 'Ext.data.Model',
fields: [
{ name: 'id', type: 'int' },
{ name: 'number', type: 'int', allowNull: true },
{ name: 'state', type: 'auto' }
]
});
```

#### Note

Sometimes you may have a type `int` property in your model and it could be `null`. Sencha, by default, assigns the value '0' (zero) to this property, so if you do not fill this property and you send to the server the model, you will send a zero value in this property instead of `null` value. The solution for this is add `allowNull` property to the `int` model property with `true` value. Example:

```
{ name: 'number', type: 'int', allowNull: true }
```

### 3.38.5 Step 4: Create the Store

`Ext.data.Store` can be thought of as a collection of records, or `Ext.data.Model` instances.

Create the file `Tables` in `app/store/table` folder:

```
Ext.define('demo.store.table.TableS', {
extend: 'Ext.data.Store',
requires: ['demo.model.table.TableM'],
model: 'demo.model.table.TableM',
alias: 'store.table',
storeId: 'miStore',
autoLoad: true,
proxy: {
type: 'rest',
url: 'http://localhost:8081/devonfw-sample-server/services/rest/tablemanagement/v1/table',
withCredentials: true
}
});
```

We have to include in `Application.js` the new Store we have just created:

```
Ext.define('demo.Application', {
extend: 'Devon.App',
```

```
controllers: [
  'demo.controller.main.MainController',
  'demo.controller.page1.Page1Controller',
  'demo.controller.table.TableController'
],
name: 'demo',
stores: [
  'demo.store.table.Tables'
],
launch: function () {
  // TODO - Launch the application
}
});
```

### 3.38.6 Step 5: Create the view and viewController

We are going to start by creating the view that lists the tables.

The first step is to create a `TableListV.js` in the path `app/view/table/` that contains the reference to the `ViewController`.

```
Ext.define("demo.view.table.TableListV", {
  extend: "Ext.panel.Panel",
  alias: 'widget.tables',
  /*view requires*/
  requires: [
    'Ext.grid.Panel',
    'demo.view.table.TableListVC'
  ],
  title: i18n.tables.title,
  /*View controller reference*/
  controller: "tablelist",
  closable: true,
  initComponent: function() {
    Ext.apply(this, {
      items : [
        this.grid()
      ]
    });
    this.callParent(arguments);
  },
  grid : function() {
    return {
      xtype: 'grid',
      reference: 'tablesgrid',
      flex: 1,
      padding: '0 10 10 10',
      allowDeselect: true,
      store:this.getStore(),
      columns: [
        {
          text: i18n.tables.grid.number,
          dataIndex: 'number'
        },
        {
          text: i18n.tables.grid.state,
          dataIndex: 'state',
          flex: 1
        }
      ]
    },
    getStore:function(){
      return Ext.create('store.table', {name:'storetable_'+Math.random()});
    }
  });
});
```

Now, the ViewController. Create the file `TableListVC.js` in the folder `app/view/table/`:

```
Ext.define('demo.view.table.TableListVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.tablelist'
});
```

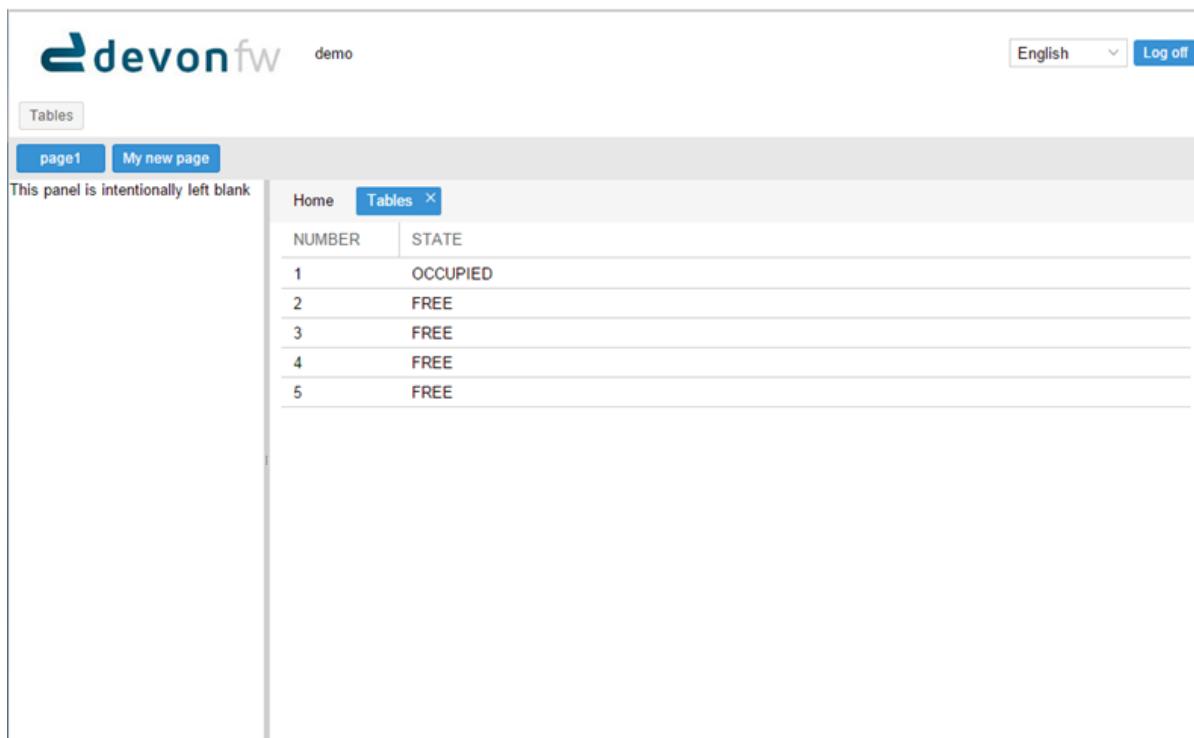
### 3.38.7 Step 6: Create i18n literals

In order to properly have the application internationalized, it is mandatory to define the bundle of messages for each language to support.

In the different views, we have defined the texts, in function of the value of some properties defined in a special `i18n` object. For every group of views (in this case, tables views), we need to create another file called `Table_<language code>_<country code>.js`. So we are going to create a file `Table_en_EN.js` in the path `app/view/table/i18n/`.

```
Ext.define('demo.view.table.i18n.Table_en_EN', {
    extend: 'Devon.I18nBundle',
    singleton: true,
    i18n: {
        tables: {
            title: 'Tables',
            html: 'List of tables for the restaurant demo',
            grid: {
                number: 'NUMBER',
                state: 'STATE'
            }
        }
    }
});
```

Navigate to our application in the browser and check the changes we have just made. The result should be like this one:



NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE
4	FREE
5	FREE

### 3.39 Complete CRUD example (Create, Read, Update and Delete)

In order to complete our example, we are going to add to our page several operations we can do with a Table (Create table, Update table, Read tables and Delete tables).

First of all, we define the following properties in the grid in `TableListV.js` so that we add all the needed buttons to invoke the new business operations.

```
grid: function() {
  return {
    xtype: 'grid',
    reference: 'tablesgrid',
    flex: 1,
    padding: '0 10 10 10',
    allowDeselect: true,
    store: this.getStore(),
    columns: [<{
      text: i18n.tables.grid.number,
      dataIndex: 'number'
    }, {
      text: i18n.tables.grid.state,
      dataIndex: 'state',
      flex: 1
    }],
    tbar: {
      items: [<{
        text: i18n.tables.buttons.add,
        handler: 'onAddClick'
      }, {
        text: i18n.tables.buttons.edit,
        handler: 'onEditClick'
      }, {
        text: i18n.tables.buttons.del,
        handler: 'onDeleteClick'
      }]
    }
  }
},
```

Then, we have to change the `TableListVC.js` to add the functionality that we need for our view:

```
Ext.define('demo.view.table.TableListVC', {
  extend: 'Ext.app.ViewController',
  alias: 'controller.tablelist',

  onAddClick: function() {
    Ext.GlobalEvents.fireEvent('eventTableAdd');
  },
  onEditClick: function() {
    var grid = this.lookupReference('tablesgrid');
    var rowSelected = grid.getSelectionModel().selected.items[0];
    var id = rowSelected.data.id;
    Ext.GlobalEvents.fireEvent('eventTableEdit', id);
  },
  onDeleteClick: function() {
    var me = this;
    Ext.MessageBox.confirm('Confirmar', i18n.tables.buttons.deleteMsg,
      function(buttonPressed) {
        if (buttonPressed == 'no' || buttonPressed == 'cancel') {
          return;
        }
        var grid = me.lookupReference('tablesgrid');
        var rowSelected = grid.getSelectionModel().selected.items[0];
        Devon.rest.tablemanagement.table.del({
          id: id
        });
      }
    );
  }
});
```

```
        scope: me,
        withCredentials: true,
        uriParams: {
            id: rowSelected.get('id')
        },
        success: me.refreshGrid
    }) ;
}
);
};

refreshGrid: function() {
    var grid = this.lookupReference('tablesgrid');
    grid.getStore().reload();
}
});
```

We have to change `TableController.js` in order to add the listeners for the events we have defined in the file `TableListVC.js`:

```

Ext.define('demo.controller.table.TableController', {
    extend: 'Ext.app.Controller',
    /*view requires*/
    requires:[
        'demo.view.table.i18n.Table_en_EN',
        'demo.view.table.TableListV',
        'demo.view.table.TableEditV'
    ],
    /*Global events listeners definition*/
    config: {
        listen: {
            global: {
                eventOpenTableList: 'onMenuOpenTables',
                eventTableAdd: 'onTableAdd',
                eventTableEdit: 'onTableEdit'
            }
        }
    }
},
/*Rest end points definition*/
init: function() {
    Devon.Ajax.define({
        'tablemanagement.table': {
            url: 'tablemanagement/v1/table/{id}'
        }
    });
},
/*Create a view when a global event is fired*/
onMenuOpenTables: function() {
    var tables = new demo.view.table.TableListV();
    Devon.App.openInContentPanel(tables);
},
//We use window for add case to show an example of how to work with window
onTableAdd: function() {
    this.openAddEditWindow();
},
onTableEdit: function(id) {
    this.openAddEditWindow(id);
},
openAddEditWindow: function(idValue){
    var title = idValue ? 'Edit Table' : 'New Table';
    var window = Ext.create('Ext.window.Window', {
        title: title,
        width: 400,
        layout: 'fit',
        closable:false,
        draggable:true,
        ...
    });
}
}

```

```

        resizable:false,
        modal:true,
        items: [
            xtype:'tableedit',
            params: {'id' : idValue}
        ],
        listeners: {
            scope: this,
            eventDone: 'closeWindow'
        }
    ).show();
},
closeWindow: function(window){
    window.close();
}
);

```

After defining the Controller, we have to create the window for the addition and edition of the tables. Create the page TableEditV.js in app/view/table:

```

Ext.define("demo.view.table.TableEditV", {
    extend: "Ext.panel.Panel",
    alias: 'widget.tableedit',
    requires: [
        'Ext.grid.Panel',
        'demo.view.table.TableEditVC'
    ],
    controller: "table-edit-controller",
    initComponent: function() {
        Ext.apply(this, {
            items : [
                this.formpanel()
            ]
        });
        this.callParent(arguments);
    },
    formpanel : function(){
        return {
            xtype:'form',
            reference:'panel',
            defaults:{ margin : 5 },
            items : [
                {
                    xtype:'hiddenfield',
                    reference:'id',
                    name: 'id'
                },
                {
                    xtype:'numberfield',
                    reference:'number',
                    fieldLabel:i18n.tableEdit.number,
                    tabIndex:1,
                    minValue:1,
                    name: 'number'
                },
                {
                    xtype:'combo',
                    reference:'state',
                    fieldLabel:i18n.tableEdit.state,
                    tabIndex:2,
                    queryMode: 'local',
                    displayField: 'code',
                    valueField: 'code',
                    name: 'state',
                    store: this.getStore()
                },
                {
                    xtype:'hiddenfield',
                    reference:'modificationCounter',
                    name: 'modificationCounter'
                }
            ],
            bbar: [

```

```

        ' -> , {
            text: i18n.tableEdit.submit,
            handler: 'onTableEditSubmit'
        } , {
            text: i18n.tableEdit.cancel,
            handler: 'onTableEditCancel'
        }
    ]
},
}

getStore: function(){
    return Ext.create('Ext.data.Store', {
        fields: ['code'],
        data:[
            {'code':'FREE'},
            {'code':'OCCUPIED'},
            {'code':'RESERVED'}
        ]
    });
}
);
}
});
```

As we have created a view, we will need to create the ViewController related to the view.

This View Controller is defining the actions to perform when the submit or cancel button is pressed. Also, it is responsible for getting the data for a table if it is an edit operation.

Create the file TableEditVC.js in the path app/view/table/:

```

Ext.define('demo.view.table.TableEditVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.table-edit-controller',
    control: {
        '#': {
            afterrender: 'onAfterRender'
        }
    },
    onTableEditSubmit: function() {
        var form = this.lookupReference('panel');
        var params = form.getValues();
        Devon.rest.tablemanagement.table.post({
            scope: this,
            jsonData : params,
            success: function(){
                //Fire close event
                var parent = this.getView().up();

                //If window we fire event
                if(parent.xtype=='window'){
                    parent.fireEvent('eventDone', parent);
                }
                //Iftabpanel, we close the tab
                else{
                    this.getView().close();
                }
            }
        });
    },
    onTableEditCancel: function() {
        this.tableEditClose();
    },
    onAfterRender: function(view) {
        var parentParams = view.params || {};
        if(parentParams.id){
            var form = this.lookupReference('panel');

            Devon.rest.tablemanagement.table.get({
                scope: this
            });
        }
    }
});

```

```

        uriParams: {
            id: parentParams.id
        },
        success: function(result, options){
            var formId = this.lookupReference('id');
            formId.setValue(result.id);
            var formNumber = this.lookupReference('number');
            formNumber.setValue(result.number);
            var formState = this.lookupReference('state');
            formState.setValue(result.state);
            var formModificationCounter = this.lookupReference('modificationCounter');
            formModificationCounter.setValue(result.modificationCounter);
        }
    });
},
tableEditClose: function() {
    var parent = this.getView().up();
    //If window we fire event
    if(parent	xtype=='window'){
        parent.fireEvent('eventDone', parent);
    }
}
);
}
);

```

After this, we have to complete our bundle of messages for completing the CRUD. So, we edit `demo.view.table.i18n.Table_en_EN`:

```

Ext.define('demo.view.table.i18n.Table_en_EN', {
    extend: 'Devon.I18nBundle',
    singleton: true,
    i18n: {
        tables: {
            title: 'Tables',
            html: 'List of tables for the restaurant demo',
            grid: {
                number: 'NUMBER',
                state: 'STATE'
            },
            buttons: {
                add: 'Add',
                edit: 'Edit',
                del: 'Delete',
                deleteMsg: 'Are you sure you want to delete this Table?'
            }
        },
        tableEdit: {
            number: 'Number',
            state: 'State',
            submit: 'Submit',
            cancel: 'Cancel'
        }
    }
});

```

Finally, we should edit our `TableM.js` to add a new field `modificationCounter`. This is because the business operation needs this field to find out if it is a new value or an updated one. This field gives us information about how many times a record has been modified:

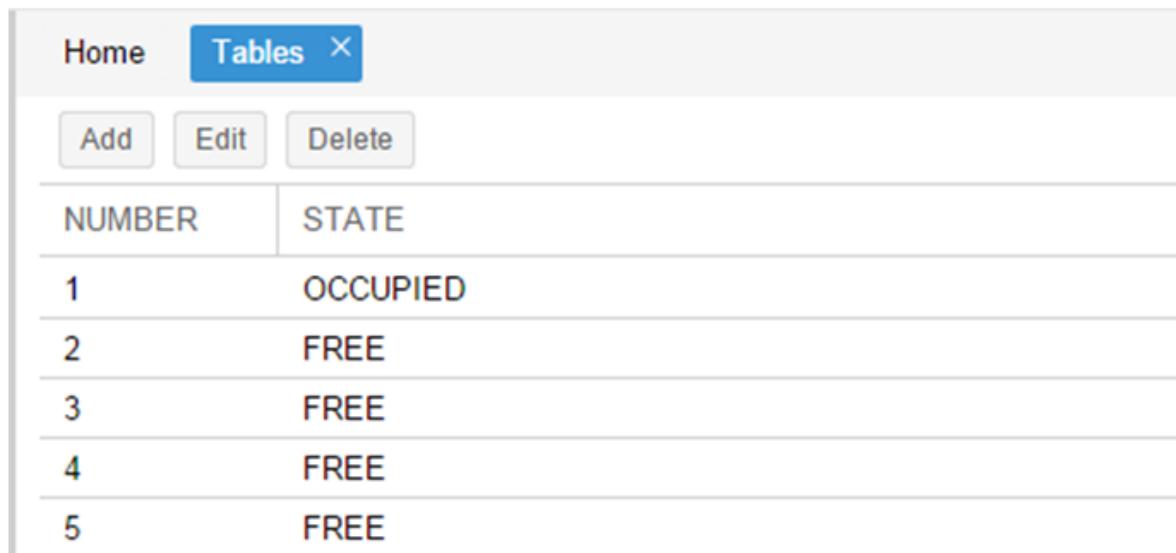
```

Ext.define('demo.model.table.TableM', {
    extend: 'Ext.data.Model',
    fields: [
        { name: 'id', type: 'int' },
        { name: 'number', type: 'int', allowNull: true },
        { name: 'state', type: 'auto' },
        { name: 'modificationCounter', type: 'int', allowNull: true}
    ]
});

```

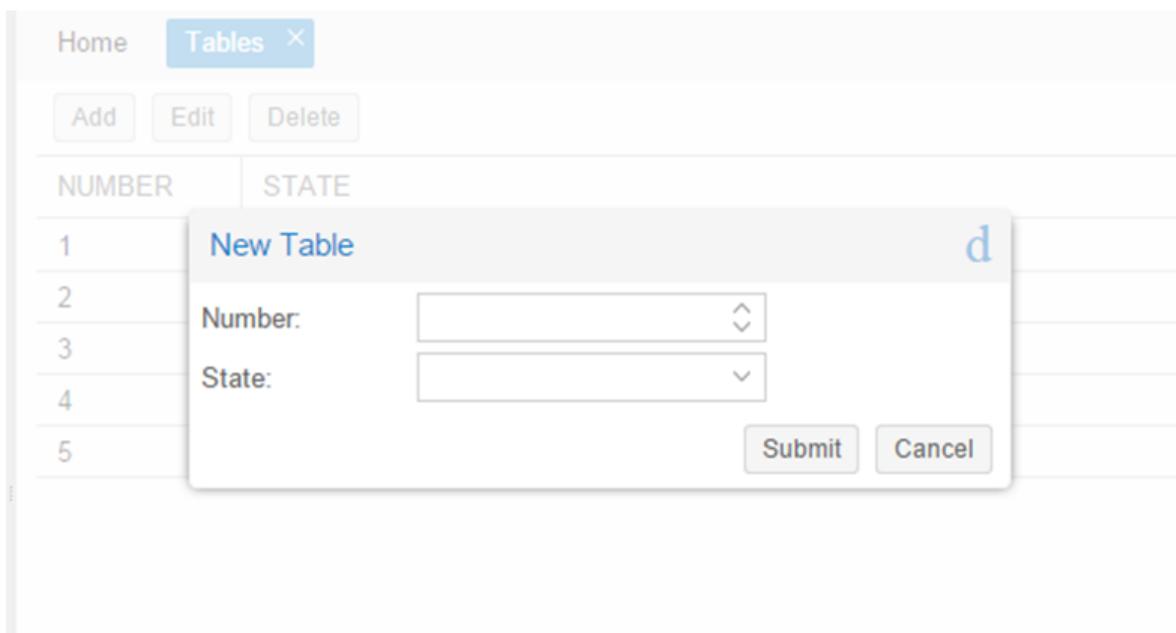
Navigate to our application in the browser and check the result of the changes

Now, we can see that we have three new buttons above our grid:



NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE
4	FREE
5	FREE

If we click the **Add** button, the application will open a new window:



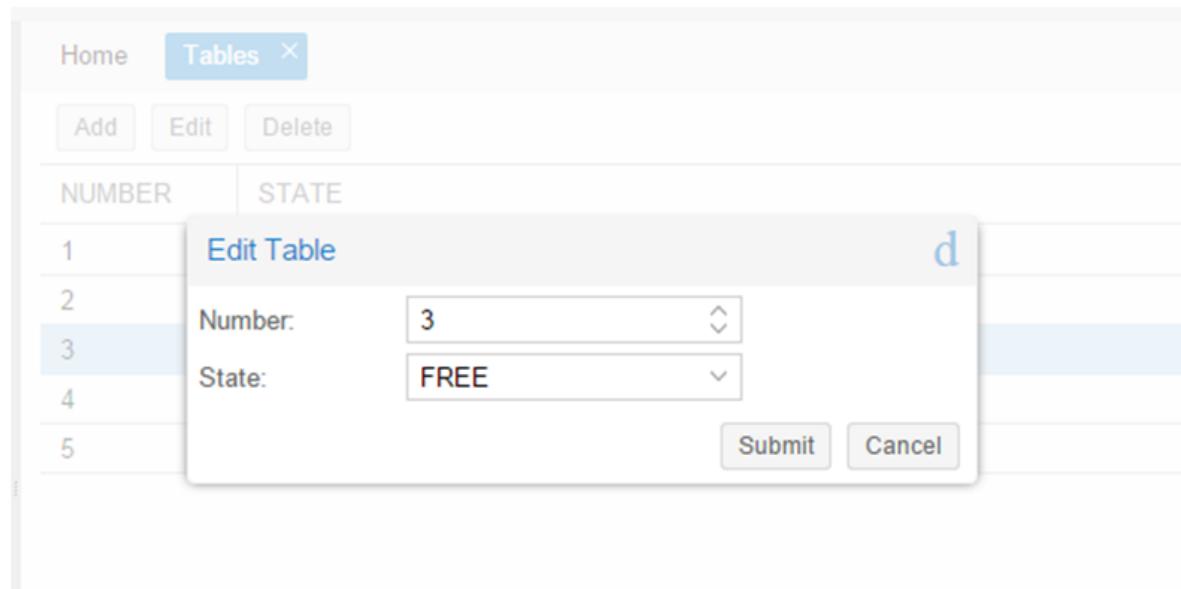
New Table

Number:

State:

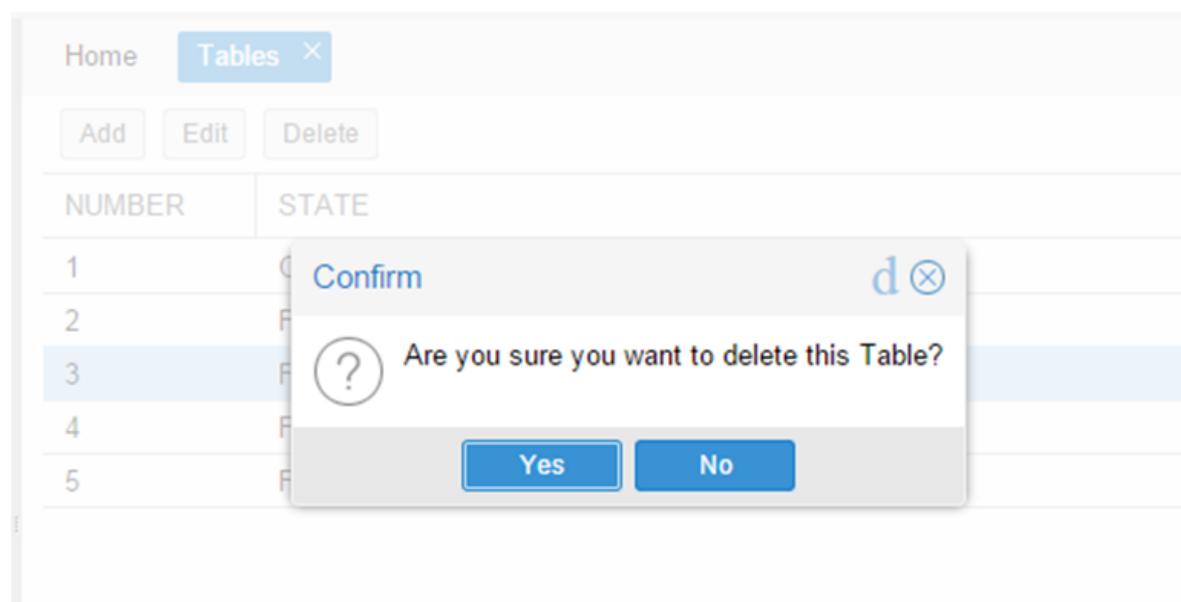
Submit Cancel

If we select a record from the grid and we click the **Edit** button:



NUMBER	STATE
1	
2	
3	
4	
5	

If we select a record from the grid and we click the Delete button:



NUMBER	STATE
1	
2	
3	
4	
5	

If we try to delete a record from the grid we will obtain the following error:

The screenshot shows a web application interface. At the top, there are navigation links: 'Home' (red), 'Tables' (blue, currently selected), and 'X'. Below this is a table grid with two columns: 'NUMBER' and 'STATE'. The data rows are numbered 1 to 5. Row 1 has 'STATE' as 'OK'. Row 2 has 'STATE' as 'FR'. Row 3 has 'STATE' as 'FR'. Row 4 has 'STATE' as 'FR'. Row 5 has 'STATE' as 'FR'. A modal dialog box is overlaid on the grid. It contains the word 'Error' in red, followed by the text 'Message: forbidden code: uuid: 3257afe3-8f70-48de-982a-153aa1673e4b'. At the bottom of the dialog is a blue 'OK' button. The background grid shows the first four rows.

This is because the user `waiter` does not have enough permission to do this functionality.

Instead of using this user, we can use the user `chief` for this purpose:

The screenshot shows a 'Login' dialog box. It has a title bar with 'Login' and a blue 'd' icon. The main area contains the text 'Please enter user and password'. Below this are two input fields: 'Username:' with the value 'chief' and 'Password:' with the value '.....'. At the bottom is a blue 'Login' button.

Use the `chief` user and try to delete a record from the tables grid

Later we will see how to disable some buttons depending on the user that is connected in that moment. With this we will be providing security to the application.

## 3.40 Extending the CRUD

In the following chapters we are going to extend the CRUD adding some functionalities.

## 3.41 Extending the CRUD

### 3.41.1 Refreshing the grid after changing some tables

When we add or edit a table, we should refresh the grid so that we can see the changes without closing the tab panel and open the grid again. For that reason, we have to create an event

eventTablesChanged in TableEditVC.js. Every time we click the submit button, window will close and we will see the grid data updated:

```
onTableEditSubmit: function() {
    var form = this.lookupReference('panel');
    var params = form.getValues();
    Devon.rest.tablemanagement.table.post({
        scope: this,
        jsonData : params,
        success: function(){
            //Fire event table changed
            Ext.GlobalEvents.fireEvent('eventTablesChanged');

            //Fire close event
            var parent = this.getView().up();
            ...
        }
    });
}
```

Then, when we click on the submit button, we will fire the eventTablesChanged event. So, we need some point in our application which captures that event. In our case, in the view-controller TableListVC.js we add this:

```
Ext.define('demo.view.table.TableListVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.tablelist',

    listen: {
        global: {
            eventTablesChanged: 'onTablesChanged'
        }
    },

    onAddClick: function() {
        Ext.GlobalEvents.fireEvent('eventTableAdd');
    },
    ...
});
```

After that, in the same file, TableListVC.js, we define the function onTablesChanged that is going to refresh the grid:

```
onTablesChanged: function() {
    this.refreshGrid();
}
```

Navigate to our application and check how the grid is refreshed after add or edit a table.

### 3.41.2 Double-click functionality

If we want to edit a table we have to select the table in the grid and then click on Edit button. We are going to do the same just with a double-click on the table.

First of all, we need a listener for the grid in TableListV.js:

```
listeners: {
    beforeitemdblclick: 'onEditDblclick'
}
```

Then, in TableListVC.js we add the function:

```
onEditDblclick: function(view, record, item, index, e, eOpts) {
    Ext.GlobalEvents.fireEvent('eventTableEdit', record.get('id'));
}
```

Navigate to our application and check how the double-click event works.



## 3.42 Extending the CRUD: ViewModels

The place to store information for a view is the ViewModel. This object contains not only instance data but also calculated fields based on other viewModel. All this information can be referenced on the view object by means of the binding capabilities of the Sencha Framework. One ViewModel object instance is associated to each View instance and they share lifespan.

We create in app/view/table the file TableListVM.js:

```
Ext.define('demo.view.table.TableListVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.table-tables',
    requires: ['demo.model.table.TableM'],
    data: {
        selectedItem: false
    }
});
```

In our TableListV.js we are going to use the selectedItem property in the grid and add the reference of the **ViewModel**:

```
requires: [
    'Ext.grid.Panel',
    'demo.view.table.TableListVM',
    'demo.view.table.TableListVC'
],  
  
viewModel: {  
    type: "table-tables"  
},  
  
...  
  
grid: function() {  
    return {  
        xtype: 'grid',  
        reference: 'tablesgrid',  
        flex: 1,  
        padding: '0 10 10 10',  
        allowDeselect: true,  
        store: this.getStore(),  
        bind: {  
            selection: '{selectedItem}'  
        },  
        columns: [{  
            text: i18n.tables.grid.number,  
            dataIndex: 'number'  
        }, {  
            text: i18n.tables.grid.state,  
            dataIndex: 'state',  
            flex: 1  
        }],  
        tbar: {  
            items: [{  
                text: i18n.tables.buttons.add,  
                handler: 'onAddClick'  
            }, {  
                text: i18n.tables.buttons.edit,  
                bind: {  
                    disabled: '{!selectedItem}'  
                },  
                handler: 'onEditClick'  
            }, {  
                text: i18n.tables.buttons.del,  
                bind: {  
                    disabled: '{!selectedItem}'  
                }  
            }]  
    }  
},
```

```
        },
        handler: 'onDeleteClick'
    }]
},
listeners: {
    beforeitemdblclick: 'onEditDblclick'
}
}
```

We can observe in the grid that if we haven't selected any record in the grid, the `Edit` and `Delete` buttons are disabled. On the contrary, if we select one record, the two buttons are enabled. With the `ViewModel` every time that the property `selectedItem` change, the value is updated in the `ViewModel`.

We also can use the property `selectedItem` to get the selected record and simplify our code. Edit the file `TableListVC.js` to use the `ViewModel`:

```
onEditClick: function() {
    var rec = this.getViewModel().get('selectedItem');
    var id = rec.data.id;
    Ext.GlobalEvents.fireEvent('eventTableEdit', id);
},
onDeleteClick: function() {
    var me = this;
    Ext.MessageBox.confirm('Confirmar', i18n.main.deleteConfirmMsg,
        function(buttonPressed) {
            if (buttonPressed == 'no' || buttonPressed == 'cancel') {
                return;
            }
            var rec = me.getViewModel().get('selectedItem');
            Devon.rest.tablemanagement.table.del({
                scope: me,
                uriParams: {
                    id: rowSelected.get('id')
                },
                success: me.refreshGrid
            });
        }
    );
},
};
```

Test the demo application. We will see that it is working as before.

The only difference, as we said before, is that when there is no record selected, the `Edit` and `Delete` buttons are disabled:

Home Tables ×

Add Edit Delete

NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE
4	FREE
5	FREE

Besides that, we can add to our ViewModel `TableListVM.js` the store we have defined for the Table entity:

```
Ext.define('demo.view.table.TableListVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.table-tables',
    requires : ['demo.model.table.TableM'],
    data: {
        selectedItem: false
    },
    stores: {
        tables: {
            model: 'demo.model.table.TableM',
            proxy: {
                type : 'tablemanagement.table'
            },
            autoLoad:true
        }
    }
});
```

As we have changed the way we define the store for the Table example, we have to use the store in the grid editing `TableListV.js` adding the binding to the store in the grid function. So, we have to remove the property `store` in the grid that we had before for this one included in the binding:

```
bind: {
    store: '{tables}',
    selection: '{selectedItem}'
},
```

As you have seen, the store that provides data to the grid has been binded to a store property in the view model.

Then, we can delete our `TableS.js` and delete the function `getStore()` in `TableListV.js`:

```
/*getStore:function(){
    return Ext.create('store.table', {name:'storetable_'+Math.random()});
}*/
```

We can see that we have changed the way we connect with our service to get the tables.

Now, we are going to use **REST endpoints**:

For easing the communication from Javascript code to the back-end, Devon provides helpers to define Rest endpoints as Javascript objects with methods to do a GET/POST/PUT/DELETE operations.

These Rest endpoints are usually created on the **Controllers**, so they get instantiated at application launch and then can be used within other Controller and **ViewController** instances.

On the sample application the **Rest endpoints** used for all the table related operations are created on the global Table Controller.

So, we have to include in our controller, `TableController.js` the definition of the operation to obtain all the tables from the server side:

```
/*Rest end points definition*/
init: function() {
    Devon.Ajax.define({
        'tablemanagement.table': {
            url: 'tablemanagement/v1/table/{id}'
        }
    });
},
```

We also have to delete in the file `Application.js` the reference to the store:

```
Ext.define('demo.Application', {
    extend: 'Devon.App',

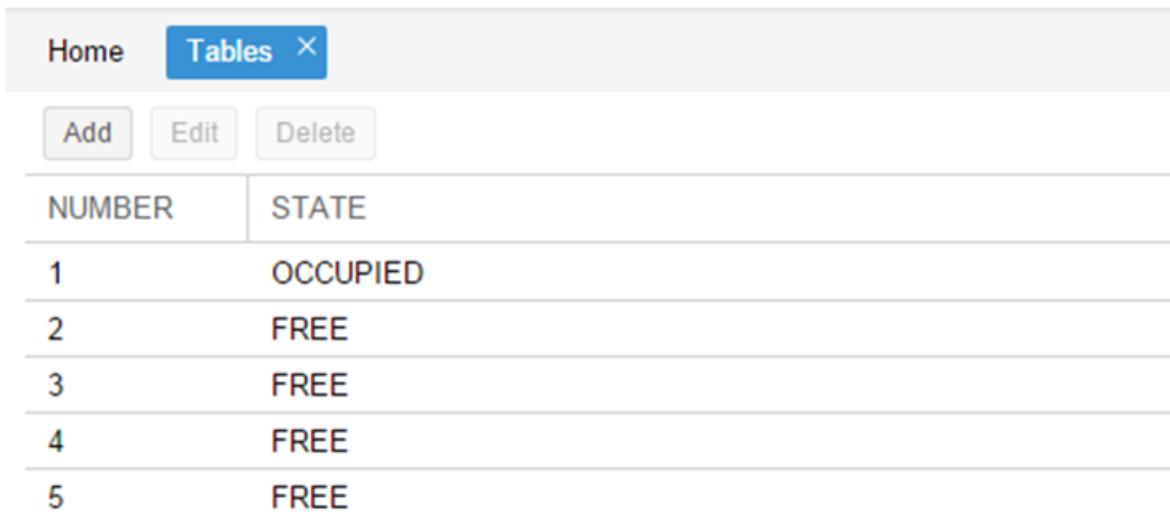
    controllers: [
        'demo.controller.main.MainController',
        'demo.controller.page1.Page1Controller',
        'demo.controller.table.TableController'
    ],

    name: 'demo',

    stores: [
    ],

    launch: function () {
        // TODO - Launch the application
    }
});
```

After all of these changes we can navigate to the grid and we can observe that the store has been loaded correctly:



NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE
4	FREE
5	FREE

Now, it is time for the edition window. So, we are going to create the `ViewModel` for this window defining the file `TableEditVM.js`:

```
Ext.define('demo.view.table.TableEditVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.table-edit-model',
    data: {
        table: {
            id: null,
            number: null,
            state: null,
            modificationCounter: null
        }
    },
    stores: {
        states: {
            fields: ['code'],
            data: [
                {'code': 'FREE'},
                {'code': 'OCCUPIED'},
                {'code': 'RESERVED'}
            ]
        }
    }
});
```

```
        ]
    }
});
```

Once we have defined the ViewModel, we are going to edit our view TableEditV.js so that we use the properties defined in the ViewModel.

First of all, we have to define the reference to the ViewModel and include it in the view in the 'requires' property. Edit the file TableEditV.js:

```
requires: [
    'Ext.grid.Panel',
    'demo.view.table.TableEditVM',
    'demo.view.table.TableEditVC'
],
controller: "table-edit-controller",
viewModel: {
    type: "table-edit-model"
},
```

After defining the ViewModel that we are going to use for this view, we have to use the properties defined in it. So, the formpanel now should be like this:

```
formpanel : function(){
    return {
        xtype:'form',
        reference:'panel',
        defaults:{ margin : 5 },
        items : [
            {
                xtype:'hiddenfield',
                reference:'id',
                name: 'id',
                bind:{
                    value:'{table.id}'
                }
            },
            {
                xtype:'numberfield',
                reference:'number',
                fieldLabel:i18n.tableEdit.number,
                tabIndex:1,
                minValue:1,
                name: 'number',
                bind:{
                    value:'{table.number}'
                }
            },
            {
                xtype:'combo',
                reference:'state',
                fieldLabel:i18n.tableEdit.state,
                tabIndex:2,
                queryMode: 'local',
                displayField: 'code',
                valueField: 'code',
                name: 'state',
                bind: {
                    store: '{states}',
                    value: '{table.state}'
                }
            },
            {
                xtype:'hiddenfield',
                reference:'modificationCounter',
                name: 'modificationCounter',
                bind:{
                    value:'{table.modificationCounter}'
                }
            }
        ],
        bbar: [
```

```

        '->', {
            text: i18n.tableEdit.submit,
            handler: 'onTableEditSubmit'
        }, {
            text: i18n.tableEdit.cancel,
            handler: 'onTableEditCancel'
        }
    ]
},
},

```

As we can see, we don't need to define the store for the field `state` in this view. We have defined the store in the ViewModel so what we do here is use that store. Then, we can delete the definition for the state store:

```

/*getStore: function(){
    return Ext.create('Ext.data.Store', {
        fields: ['code'],
        data:[
            {'code':'FREE'},
            {'code':'OCCUPIED'},
            {'code':'RESERVED'}
        ]
    });
}/*

```

Now, in the ViewController `TableEditVC.js` we can use the ViewModel to get the data of the View. Then, our `onTableEditSubmit` function when we submit the data we just added or updated should be like this:

```

onTableEditSubmit: function() {
    var vm = this.getViewModel();
    Devon.rest.tablemanagement.table.post({
        scope: this,
        jsonData : vm.get('table'),
        success: function(){
            //Fire event table changed
            Ext.GlobalEvents.fireEvent('eventTablesChanged');

            //Fire close event
            var parent = this.getView().up();

            //If window we fire event
            if(parent.xtype=='window'){
                parent.fireEvent('eventDone', parent);
            }
            //Iftabpanel, we close the tab
            else{
                this.getView().close();
            }
        }
    });
},

```

Instead of getting the data from the form, we get the data from the ViewModel. The same we do with the function `onAfterRender`:

```

onAfterRender: function(view) {
    var parentParams = view.params || {};
    if(parentParams.id){
        Devon.rest.tablemanagement.table.get({
            scope: this,
            uriParams: {
                id: parentParams.id
            },
            success: function(table){

```

```
        var vm = this.getViewModel();
        vm.set('table', table);
    }
});
```

As we can see, we get the value of the identifier of the table using the parameters of the view. Instead of doing that, we can use a ViewModel created for that purpose. So, in our `TableController.js` we have to send the parameters using a ViewModel:

```
openAddEditWindow: function(idValue){  
    var title = idValue ? 'Edit Table' : 'New Table';  
  
    var window = Ext.create('Ext.window.Window', {  
        title: title,  
        width: 400,  
        layout: 'fit',  
        closable:false,  
        draggable:true,  
        resizable:false,  
        modal:true,  
        items: [{  
            xtype:'tableedit',  
            viewModel: {  
                data: {  
                    tableId: idValue  
                }  
            }  
        }  
    }],  
    listeners: {  
        scope: this,  
        eventDone: 'closeWindow'  
    }  
}).show();  
},
```

Now, we are using a `ViewModel` even to send parameters to the window. Then, in the controller of our window `TableEditVC`, is we can instanciate the `viewModel` in order to get the data:

```
onAfterRender: function() {
    var vm = this.getViewModel();
    var id = vm.get("tableId");
    if(id){
        Devon.rest.tablemanagement.table.get({
            scope: this,
            uriParams: {
                id: id
            },
            success: function(table){
                vm.set('table', table);
            }
        });
    }
},
```

Navigate to the browser and check that everything is working as before so we can add or edit tables without any problem:

Edit Table

Number:	<input type="text" value="2"/>
State:	<input type="text" value="OCCUPIED"/>

## 3.43 Pagination

We have completed the CRUD example, but as we can see, our grid can contain a lot of records and we want to manage this amount of data in a better way. For that purpose we should paginate our grid in order to organize the information properly. Devon4sencha provides a plugin for the pagination of the grids.

To use the pagination plugin in our sample we need to:

- Define a new REST endpoint for the paginated operation in the server. We are going to modify REST endpoints definition section in `TablesController.js` adding a new endpoint with the special flag `pagination` activated:

```
init: function() {
    Devon.Ajax.define({
        'tablemanagement.table': {
            url: 'tablemanagement/v1/table/{id}'
        },
        'tablemanagement.search': {
            url: 'tablemanagement/v1/table/search',
            pagination: true
        }
    });
},
```

- Add the rest end point to the tables store in the file `TableListVM.js`:

```
stores: {
    tables: {
        model: 'demo.model.table.TableM',
        pageSize: 3,
        proxy: {
            type: 'tablemanagement.search'
        },
        autoLoad: true
    }
}
```

What we are doing here is to change the rest operation to obtain the data paginated. Also, we are defining the amount of records that we will have in each page, in this case 3.

- Add the plugin pagination to the grid. We are going to add this plugin to `TableListV.js`:

```
xtype: 'grid',
reference: 'tablegrid',
plugins: ['pagination'],
```

- Add dependencies to the List view. We need to require Devon pagination plugin. We add a dependency to `TableListV.js`:

```
requires: [
    'Ext.grid.Panel',
    'Devon.grid.plugin.Pagination',
    'demo.view.table.TableListVM',
    'demo.view.table.TableListVC'
],
```

Navigate to the demo application and check the output

The result should be like this:

NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE

« < | Page  of 2 | > » | C

If we open the folder packages\local\devon-extjs\src\grid\plugin we can see the source code for this plugin in the file Pagination.js:

```
Ext.define('Devon.grid.plugin.Pagination', {
    extend: 'Ext.plugin.Abstract',
    alias: 'plugin.pagination',

    requires: [
        'Ext.grid.Panel',
        'Ext.toolbar.Paging'
    ],

    init: function(grid) {
        var me = this;

        // If we do not find a binding store, we throw an error
        if (!grid || !grid.getInitialConfig() || !grid.getInitialConfig().bind || !grid.getInitialConfig().bind.store) {
            Devon.Log.error("Not found binding for store");
            throw ("Not found binding for store");
        }

        var pagingCfg = {
            bind: {
                store: grid.getInitialConfig().bind.store
            },
            dock: 'bottom'
        };

        grid.addDocked(Ext.create('Ext.toolbar.Paging', pagingCfg));

        me.callParent(arguments);
    }
});
```

### 3.44 Extending the CRUD: searching and filtering tables

Now we are going to add some filters to our view so that we can search and filter the records of the grid.

First of all, we have to define the search panel in our `TableListV.js`:

```
initComponent: function() {
    Ext.apply(this, {
        items : [
            this.search(),
            this.grid()
        ]
    });
    this.callParent(arguments);
},

search : function(){
    return {
        xtype:'form',
        reference:'form',
        title:'Filters',
        border:true,
        collapsible:true,
        layout:{
            type:'hbox',
            align:'top'
        },
        margin : 10,
        defaults:{ margin : 10 },
        items : [{{
            xtype:'panel',
            border:false,
            layout:{type:'vbox',align:'stretch'},
            flex:5,
            items:[{
                xtype:'numberfield',
                reference:'number',
                fieldLabel:i18n.tableEdit.number,
                bind:'{filters.number}',
                minValue:1,
                labelWidth:80,
                tabIndex:1,
                flex:1
            }]
        },{
            xtype: 'panel',
            border:false,
            layout:{type:'vbox',align:'stretch'},
            flex:5,
            items:[{
                xtype:'combo',
                reference:'state',
                fieldLabel:i18n.tableEdit.state,
                tabIndex:2,
                queryMode: 'local',
                displayField: 'code',
                valueField: 'code',
                name: 'state',
                bind: {
                    store: '{states}',
                    value: '{filters.state}'
                }
            }]
        }],
        bbar:[
            '>',
            {
                xtype:'button',
                text:'Search',
                width:80,
            }
        ]
    };
}
```

```
        listeners: {
            click: 'doSearch'
        }
    },
    xtype: 'button',
    text: 'Clean',
    width: 80,
    listeners: {
        click: 'doClean'
    }
}
];
}
};
```

We have to extend the ViewModel for the view so we edit `TableListVM.js` adding the filters. Also, we have to add the store for the state of the table we want to filter:

```

Ext.define('demo.view.table.TableListVM', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.table-tables',
    requires : ['demo.model.table.TableM'],
    data: {
        selectedItem: false,
        filters:{ 
            number: null,
            state: null
        }
    },
    stores: {
        tables: {
            model: 'demo.model.table.TableM',
            pageSize: 3,
            proxy: {
                type : 'tablemanagement.search'
            },
            autoLoad:true
        },
        states: {
            fields: ['code'],
            data:[
                {'code':'FREE'},
                {'code':'OCCUPIED'},
                {'code':'RESERVED'}
            ]
        }
    }
});

```

Navigate to the view in our browser. We can see that we have created a collapsible panel above the grid with two fields and two buttons:

NUMBER	STATE
1	OCCUPIED
2	FREE
3	FREE

Now it is time to define what we are going to do with these buttons we have just created. So we edit `TableListVC.js` to create the functions `doSearch` and `doClean`:

```
doSearch: function(){
    var grid = this.lookupReference('tablegrid');
    var store = grid.getStore();
    var form = this.lookupReference('form');

    if(!form.isValid()){
        return;
    }

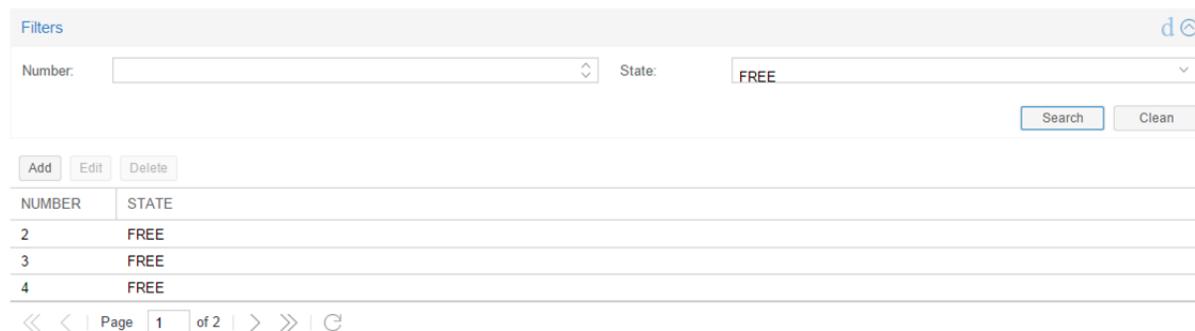
    store.load({
        params : this.getViewModel().data.filters
    });
},

doClean: function(){
    var grid = this.lookupReference('tablegrid');
    var form = this.lookupReference('form');

    grid.getStore().removeAll();
    form.getForm().reset();
}
```

Check the changes in the application.

Now, we can see that if we filter by number or state in the grid we only see the records that match with these filters.



The screenshot shows a user interface for filtering data. At the top, there is a 'Filters' section with two input fields: 'Number' and 'State'. The 'State' field contains the value 'FREE'. Below this is a table with two columns: 'NUMBER' and 'STATE'. The data in the table is:

NUMBER	STATE
2	FREE
3	FREE
4	FREE

At the bottom of the grid, there is a pagination control showing 'Page 1 of 2'.

### 3.45 Extending the CRUD: Adding an inbox to search tables

As we can see in our sample application, in our home page we have a menu in the left side of the screen. The panel is blank now but we are going to add some items in order to create a sort of inbox to search tables depending on their state (Free, Reserved or Occupied). We have to edit the file `LeftSideMenu.js` in `app/view/main/`. The content of this file should be:

```
Ext.define('demo.view.main.LeftSidePanel', {
    extend: 'Ext.Panel',
    alias: 'widget.main-leftsidepanel',
    requires: [
    ],
    cls: 'main-leftsidepanel',
    width: 220,
    bodyPadding:0,
    resizable: {
        handles: 'e',
        pinned: true
    },
    layout: {
        type:'vbox',
        align:'stretch'
    },
    items: [
        {
            xtype: 'main-res-status',
            height:220
        }
    ]
});
```

We only have added the item with the `xtype` `main-res-status`. This is a reference to a view that we are going to define now.

Create `RestaurantStatus.js` in `app/view/main/`:

```
Ext.define('demo.view.main.RestaurantStatus', {
    extend: 'Ext.tree.Panel',
    alias: 'widget.main-res-status',
    requires:[ 'demo.view.main.RestaurantStatusVC' ],
    title: i18n.restaurantStatus.title,
    collapsible:true,
    controller: 'main-res-status',
    rootVisible: false,
    root: {
        expanded: true,
        id:'root',
        children: [
            {
                text: i18n.restaurantStatus.tables,
                id:'tables',
                expanded: true,
                children: [
                    {
                        text: i18n.restaurantStatus.free,
                        id:'tables-free',
                        iconCls:'treenode-no-icon',
                        leaf: true
                    },
                    {
                        text: i18n.restaurantStatus.reserved,
                        id:'tables-reserved',
                        iconCls:'treenode-no-icon',
                        leaf: true
                    },
                    {
                        text: i18n.restaurantStatus.occupied,
                        id:'tables-occupied',
                        leaf: true
                    }
                ]
            }
        ]
});
```

```

        iconCls:'treenode-no-icon',
        leaf: true
    }]
},
});

```

In this view we are creating a treePanel to show all the states that a table can have and show the quantity of tables in each state. Then, when we click one state we want to be able to see the grid of the tables filtered by that state. For that purpose we need to create the ViewController for this view.

Create the file 'RestautantStatusVC.js' in 'app/view/main':

```

Ext.define('demo.view.main.RestaurantStatusVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.main-res-status',
    listen: {
        global: {
            eventTablesChanged: 'onTablesChanged'
        }
    },
    control: {
        '#': {
            afterrender: 'onAfterRender'
        }
    },
    onTablesChanged: function(){
        this.refreshTree();
    },
    onAfterRender: function() {
        this.refreshTree();
    },
    refreshTree: function(){
        var store=this.getView().getStore();
        Devon.rest.tablemanagement.search.post({
            jsonData:{},
            success:function(data){
                var res=data.result, free=0, occupied=0, reserved=0;
                Ext.Array.each(res,function(item){
                    if(item.state==demo.model.table.TableM.state.FREE) free++;
                    if(item.state==demo.model.table.TableM.state.OCCUPIED) occupied++;
                    if(item.state==demo.model.table.TableM.state.RESERVED) reserved++;
                });
                store.getNodeById('tables-free').set('text',i18n.restaurantStatus.free+' ('+free+')');
                store.getNodeById('tables-reserved').set('text',i18n.restaurantStatus.reserved
                +' ('+reserved+')');
                store.getNodeById('tables-occupied').set('text',i18n.restaurantStatus.occupied
                +' ('+occupied+')');
            }
        });
    }
});

```

Then, the next step is to edit the file `Main_en_EN.js` to update the messages for this next functionality:

```

restaurantStatus:{
    title:'Restaurant',
    tables: 'Tables',
    positions: 'Positions',
    free : 'Free',
    occupied : 'Occupied',
    reserved : 'Reserved',

```

```
        assigned : 'Assigned',
        total : 'Total'
    }
```

As we have created a new view in our application, we have to add the reference of this view in the 'requires' property of the `MainController.js`.

So, we edit the `MainController` to add the reference to the `RestaurantStatus` view:

```
requires: [
    'demo.view.main.i18n.Main_en_EN',
    'demo.view.main.LeftSidePanel',
    'demo.view.main.Content',
    'demo.view.main.Home',
    'demo.view.NewPage',
    'demo.view.main.RestaurantStatus'
]
```

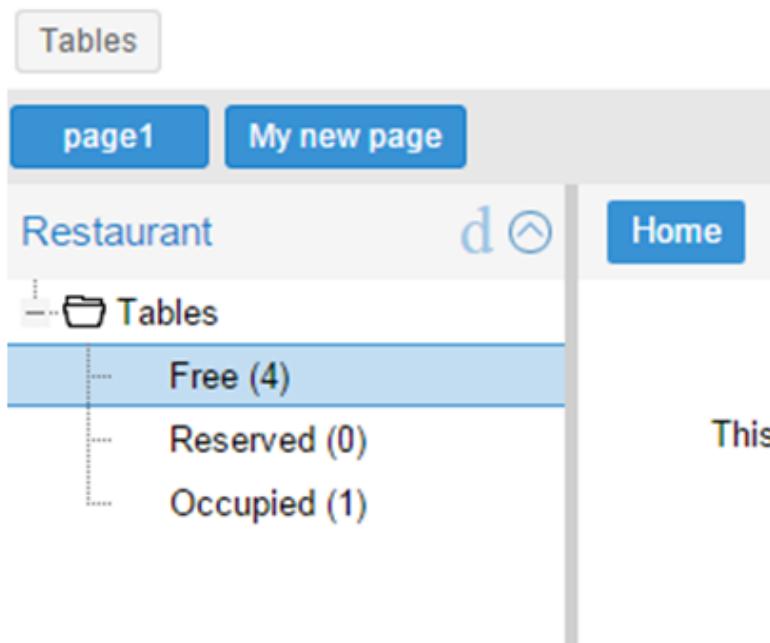
The next step is to edit the file `TableM.js` to add some static text about the state of a table. We should use an independent file for this purpose in order to have all of the constants of the application in the same file. However, in our sample we are going to use the model to keep these constants:

```
Ext.define('demo.model.table.TableM', {
    extend: 'Ext.data.Model',

    statics: {
        state: {
            OCCUPIED: 'OCCUPIED',
            FREE: 'FREE',
            RESERVED: 'RESERVED'
        }
    },
    fields: [
        { name: 'id', type: 'int' },
        { name: 'number', type: 'int', allowNull: true },
        { name: 'state', type: 'auto' },
        { name: 'modificationCounter', type: 'int', allowNull: true}
    ]
});
```

These constants are used in `RestaurantStatusVC.js` to compare the state of a table and the text of this state that we have as a constant in the model.

Navigate to our application in the browser. We will see a tree named `Tables` and three possible states: Free, Reserved or Occupied. Besides, we can see the number of tables in each state.



Now, we want to add some functionality to be able to show the table view with the grid filtered by the state selected. For achieve that, we can to edit the file `RestaurantStatus.js` adding the listener to fire the event when a node of the tree is selected:

```
listeners: {
    select: 'onSelect'
}
```

As we have added the event we have to edit the file `RestaurantStatusVC.js` to capture that event:

```
onSelect : function(tree, record){
    if(record.isLeaf()){
        var nodeId=record.getId();
        if(Ext.String.startsWith(nodeId,'table')){
            var state=nodeId.split('-')[1];
            var title=i18n.tables.title+' '+i18n.restaurantStatus[state];
            state=demo.model.table.TableM.state[state.toUpperCase()];
            Ext.GlobalEvents.fireEvent('eventOpenTableList',{title:title,stateFilter:state});
        }
    }
}
```

We have added the function `onSelect` to open the tab panel with the management of the tables but depending on the filter selected. In this function we fire the event `eventOpenTableList` which is captured by the controller `TableController.js`.

Then, we have to edit the controller to give the view the options we have just sent. These options are the title of the view and the filter of the state. Edit the function `onMenuOpenTables`:

```
onMenuOpenTables: function(options) {
    var tables = new demo.view.table.TableListV(options);
    Devon.App.openInContentPanel(tables);
},
```

Now, we are providing these configuration options to the view. The next step is to use these options when the view is rendered. In order to do that we have to define the event in the controller of the view, `TableListVC.js`:

```
listen: {
  global: {
    eventTablesChanged: 'onTablesChanged'
  },
  component: {
    'tables': {
      'afterrender': 'onAfterRender'
    }
  }
},
onAfterRender: function(panel){
  var stateFilter=this.getView().stateFilter;
  if(stateFilter) this.getViewModel().set('stateFilter',{state:stateFilter});
  else this.getViewModel().set('stateFilter',{state:null});
},
```

As we can see, we are using the ViewModel to set the state filter. Then, we have to add to our request the filter to get the data and load the store.

Edit the file TableListVM.js to add the state param:

```
tables: {
  model: 'demo.model.table.TableM',
  pageSize: 3,
  proxy: {
    type: 'tablemanagement.search',
    extraParams:'{stateFilter}'
  },
  autoLoad: true,
  remoteSort:true,
  remoteFilter:true,
  sorters: {property:'number', direction:'ASC'}
},
```

Besides adding the `extraParams` property to the proxy request, we have configured other properties:

- **remoteSort**: When we are sorting the information of the columns of the grid, we only sort the information for the page we are in that moment. In order to sort the grid by the column selected independently from the page, we have to change this property to true. With this, when the request to the backend is done it will sort the information there, not in the view according to the page.
- **remoteFilter**: Indicates if the filters are going to be done on the server or on the client. We need this property to filter the whole store, not only the data that is showed in the current page.
- **sorters**: Show the information sorted by defect with the column and direction indicated.

Check the output and use the tree panel to filter tables

Restaurant 

Tables 

- Free (4)
- Reserved (0)
- Occupied (1)

Home Tables Reserved  Tables Occupied  Tables Free 

Filters

Number:

Add Edit Delete

NUMBER ↑	STATE
2	FREE
3	FREE
4	FREE

« < | Page  of 2 | > » | C

### 3.46 Extending the CRUD: Change the state of a table

We are going to extend our sample application changing the state of the tables. For that purpose we have to add some buttons to be able to do the actions of changing between states.

First of all, we have to edit the file `TableListV.js` to add these buttons:

```

, {
  text: i18n.tables.buttons.reserve,
  handler: 'onMarkAsReserved',
  bind: {
    disabled: '{!canReserve}'
  }
}, {
  text: i18n.tables.buttons.cancel,
  handler: 'onCancelReserve',
  bind: {
    disabled: '{!canCancel}'
  }
}, {
  text: i18n.tables.buttons.occupy,
  handler: 'onMarkAsOccupied',
  bind: {
    disabled: '{!canOccupy}'
  }
}, {
  text: i18n.tables.buttons.free,
  handler: 'onMarkAsFree',
  bind: {
    disabled: '{!canFree}'
  }
}
}

```

We have added new buttons to the grid. Now, we have to add to the file `Table_en_EN.js` the messages for these buttons:

```

buttons: {
  add: 'Add',
  edit: 'Edit',
  editOrder: 'Edit Order',
  del: 'Delete',
  deleteMsg: 'Are you sure you want to delete this Table?',
  refresh: 'Refresh',
  reserve: 'Reserve',
  cancel: 'Cancel Reservation',
  occupy: 'Occupy',
  free: 'Free'
}

```

As we have seen before in our View, we will be able to find out if a button have to be enabled or not depending on the information of our ViewModel. Edit the file `TableListVM.js` to add some formulas to our ViewModel:

```

formulas: {
  canReserve: function(get) {
    var table = get('selectedItem');
    if (!table) {
      return false;
    }
    return table.get("state") == demo.model.table.TableM.state.FREE;
  },
  canCancel: function(get) {
    var table = get('selectedItem');
    if (!table) {
      return false;
    }
    return table.get("state") == demo.model.table.TableM.state.RESERVED;
  }
}

```

```

        return false;
    }

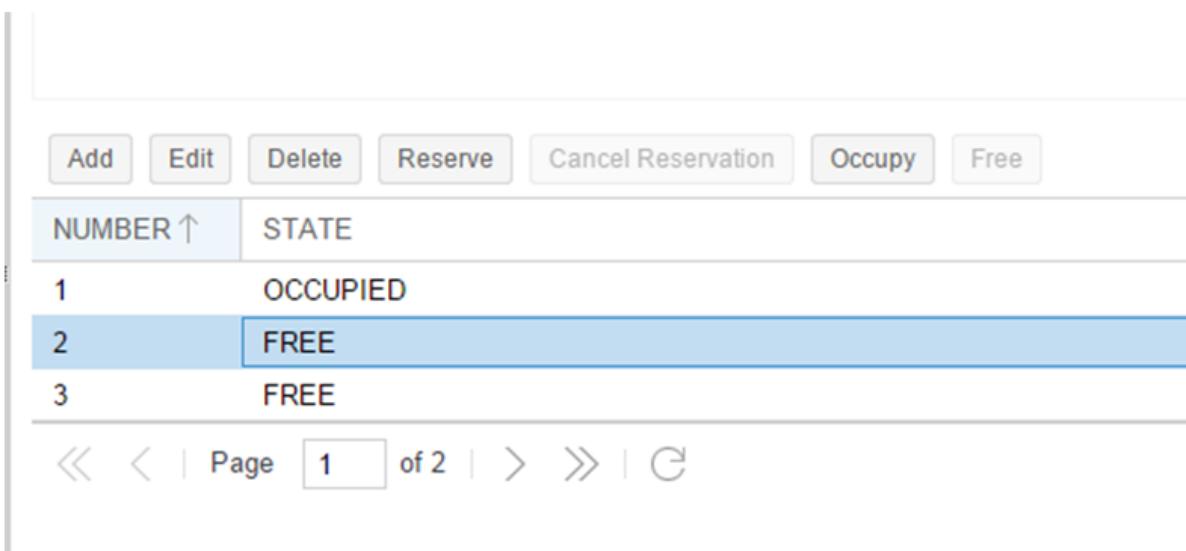
    return table.get("state") == demo.model.table.TableM.state.RESERVED;
},
canOcuppy: function(get) {
    var table = get('selectedItem');
    if (!table) {
        return false;
    }
    var state = table.get("state");
    return state == demo.model.table.TableM.state.RESERVED || state ==
demo.model.table.TableM.state.FREE;
},
canFree: function(get) {
    var table = get('selectedItem');
    if (!table) {
        return false;
    }
    return table.get("state") == demo.model.table.TableM.state.OCCUPIED;
}

}

```

Navigate to our application and check the changes

Now, if we check our sample, we will see that when we select a record from the grid the buttons enabled and disabled depending on the logic we have just added.



NUMBER ↑	STATE
1	OCCUPIED
2	FREE
3	FREE

The last step is to add the functions of these buttons in the file TableListVC.js:

```

markSelectedAs: function(status) {
    var me = this;
    var table = me.getViewModel().get('selectedItem').data;
    table.state = status;

    Devon.rest.tablemanagement.table.post({
        scope: me,
        jsonData: table,
        success: function(){
            Ext.GlobalEvents.fireEvent('eventTablesChanged');
        }
    });
},

```

```
onMarkAsOccupied: function() {
    this.markSelectedAs(demo.model.table.TableM.state.OCCUPIED);
},
onMarkAsFree: function() {
    this.markSelectedAs(demo.model.table.TableM.state.FREE);
},
onMarkAsReserved: function() {
    this.markSelectedAs(demo.model.table.TableM.state.RESERVED);
},
onCancelReserve: function() {
    this.markSelectedAs(demo.model.table.TableM.state.FREE);
}
```

Each time one of these buttons is pressed, the state of the table will change depending on the previous state.

Navigate to the application and change the state of the tables

## 3.47 Extending the CRUD: Editing menu order

### 3.47.1 Create Edit Order View

In this section we are going to order some dishes from a menu for the tables that we have in our system. For this purpose we add a new button in the grid named `Edit Order`. When we click this button it opens a new tab panel showing two grids, one with the menu information and the other one with the dishes ordered for the table.

Add the button `Edit Order` editing the file `TableListV.js`:

```
{  
    text: i18n.tables.buttons.editOrder,  
    bind: {  
        disabled: '{!selectedItem}'  
    },  
    handler: 'onEditOrderClick'  
}
```

Edit the file `TableListVC.js` to add the function `onEditOrderClick`:

```
onEditOrderClick: function() {  
    var rec = this.getViewModel().get('selectedItem');  
    Ext.GlobalEvents.fireEvent('eventTableEditOrder', {  
        id: rec.id  
    });  
}
```

Edit the file `TableController` to add the listener for the event `eventTableEditOrder`:

```
config: {  
    listen: {  
        global: {  
            eventOpenTableList: 'onMenuOpenTables',  
            eventTableAdd: 'onTableAdd',  
            eventTableEdit: 'onTableEdit',  
            eventTableEditOrder: 'onTableEditOrder'  
        }  
    }  
},
```

Then, in the same file, add the definition of the function:

```
onTableEditOrder: function(tableSelected) {  
    var id = tableSelected.id;  
    var panel = new demo.view.table.TableOrderV({  
        title: i18n.tableOrder.title + id,  
        viewModel: {  
            data: {  
                tableId: id  
            }  
        }  
    });  
    Devon.App.openInContentPanel(panel, {  
        id: id  
    });  
},
```

Create the **View** with the file `TableOrderV.js` in the path `app/view/table`:

```
Ext.define("demo.view.table.TableOrderV", {  
    extend: "Ext.panel.Panel",  
    alias: 'widget.tableorder',
```

```
requires: [
    'demo.view.table.TableOrderVM',
    'demo.view.table.TableOrderVC'
],  
  
controller: "table-order-controller",  
  
viewModel: {
    type: "table-order-model"
},  
  
closable: true,  
  
initComponent: function() {
    Ext.apply(this, {
        items : [
            this.header(),
            this.gridMenu(),
            this.gridOrder()
        ]
    });
    this.callParent(arguments);
},  
  
header: function{
    return {
        padding: 10,
        bind: {
            html: i18n.tableOrder.html + '{tableId}'
        },
        border: false
    }
},  
  
gridMenu: function{
    return {
        xtype: 'grid',
        title: 'Menu',
        reference: 'menugrid',
        allowDeselect: true,
        columns: [
            {
                text: i18n.tableOrder.grid.number,
                dataIndex: 'id'
            },
            {
                text: i18n.tableOrder.grid.title,
                dataIndex: 'description',
                flex: 1
            },
            {
                text: i18n.tableOrder.grid.price,
                dataIndex: 'price'
            }
        ],
        bind: {
            store: '{menu}',
            selection: '{selectedMenuItem}'
        },
        bbar: {
            items: [
                {
                    text: i18n.tableOrder.add,
                    bind: {
                        disabled: '{!selectedMenuItem}'
                    },
                    handler: 'addOrder'
                }
            ]
        }
    }
},  
  
gridOrder: function{
    return {
        xtype: 'grid',
```

```

        title: 'Orders',
        reference: 'ordergrid',
        allowDeselect: true,
        columns: [{
            dataIndex: 'id',
            hidden: true
        }, {
            text: i18n.tableOrder.grid.title,
            dataIndex: 'description',
            flex: 1
        }, {
            text: i18n.tableOrder.grid.price,
            dataIndex: 'price'
        }],
        bind: {
            store: '{order}',
            selection: '{selectedOrderItem}'
        },
        bbar: [
            items: [
                text: i18n.tableOrder.remove,
                bind: {
                    disabled: '{!selectedOrderItem}'
                },
                handler: 'orderRemove'
            },
            '->', {
                text: i18n.tableOrder.submit,
                handler: 'tableOrderSubmit'
            },
            {
                text: i18n.tableOrder.cancel,
                handler: 'tableOrderCancel'
            }
        ]
    }
}
});
```

Create the **ViewModel** with the file TableOrderVM.js:

```

Ext.define('demo.view.table.TableOrderVM', {
    extend: 'Ext.app.ViewModel',
    requires: [
    ],
    alias: 'viewmodel.table-order-model',

    data: {
        selectedMenuItem: false,
        selectedOrderItem: false
    },

    stores: {
        menu: {
            fields: ['id', 'description', 'price'],
            data:[
                {'id': 1, 'description':'Coke', 'price':'3.99'},
                {'id': 2, 'description':'Water', 'price':'1.99'},
                {'id': 3, 'description':'Orange Juice', 'price':'4.99'},
                {'id': 4, 'description':'Salad', 'price':'7.99'},
                {'id': 5, 'description':'Chicken', 'price':'8.99'}
            ]
        },
        order: {
            fields: ['id', 'offerId', 'description', 'price'],
            data:[
            ]
        }
    }
})
```

```
});
```

Create the **ViewController** with the file TableOrderVC.js:

```
Ext.define('demo.view.table.TableOrderVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.table-order-controller',

    control: {
        'tableorder': {
            afterrender: 'onAfterRender'
        }
    },

    onAfterRender: function() {
        /* */
    },

    tableOrderCancel: function() {
        this.tableOrderClose();
    },

    tableOrderClose: function() {
        this.getView().destroy();
    },

    tableOrderSubmit: function() {
        /* TODO Submit Orders for the table */
    },

    orderRemove: function() {
        var model = this.getViewModel();
        var orders = model.get("order");
        var selectedItem = model.get("selectedOrderItem");

        orders.remove(selectedItem);
    },

    addOrder: function() {
        var vm = this.getViewModel();
        var selectedMenu = vm.get("selectedMenuItem");

        vm.get("order").add({
            id: null,
            offerId: selectedMenu.get("id"),
            description: selectedMenu.get("description"),
            price: selectedMenu.get("price")
        });
    }
});
```

Edit the file Table\_en\_EN.js to add the new messages:

```
tableOrder: {
    title: 'Table: ',
    newTitle: 'New table',
    status: 'STATUS',
    orderPos: 'Order Positions:',
    add: 'Add',
    remove: 'Remove',
    submit: 'Submit',
    cancel: 'Cancel',
    html: 'Details for table #',
    grid: {
        number: 'Number',
        title: 'Title',
        status: 'STATUS',
        price: 'Price',
        comment: 'Comment'
    }
}
```

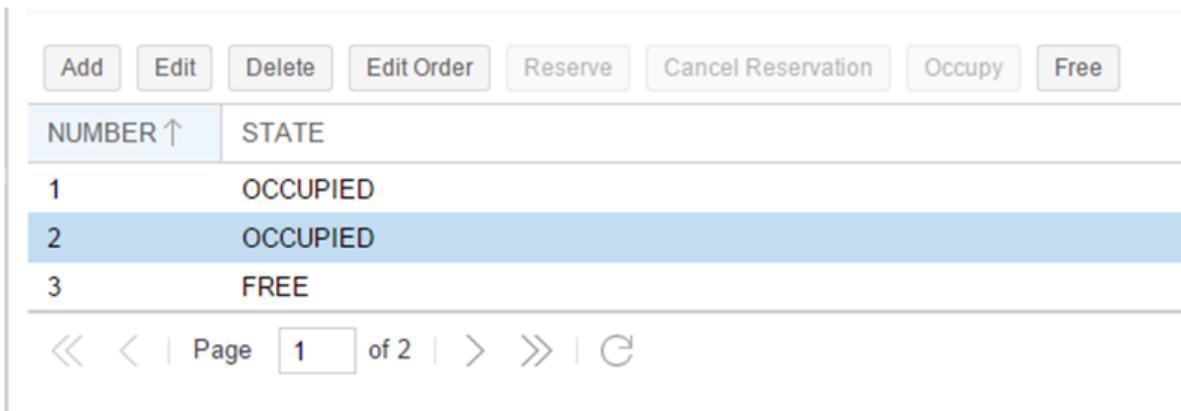
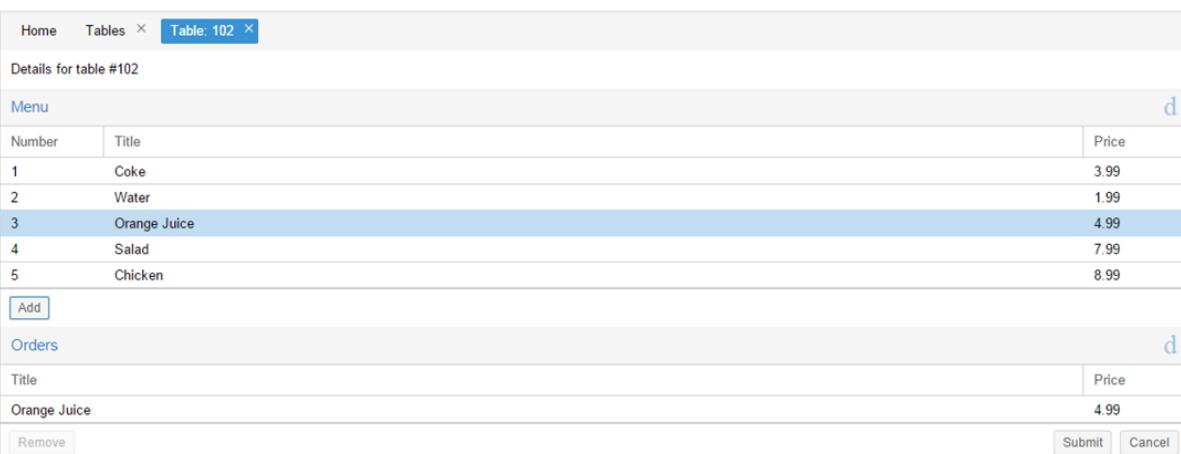
```
  },
},
```

Edit the file `TableController.js` adding the new view in the `requires` property:

```
requires:[
  'demo.view.table.i18n.Table_en_EN',
  'demo.view.table.TableListV',
  'demo.view.table.TableEditV',
  'demo.view.table.TableOrderV'
],
```

Navigate to the application and check the result

After all these steps if we check the result on the browser we will see two grids, one with the information about the menu and the other one with the information about what we have ordered for that table. When we select one item from the `Menu` grid we can add that item on the `Orders` grid so we can complete our order.

### 3.47.2 Drag and drop

We are going to change our sample a bit. Instead of using the button `Add` to add some dish or drink to our order, we are going to do it using the `Drag and drop` functionality that ExtJS provides.

The only thing we have to do is to define in our grids the configuration for the drag and drop.

Include in the first grid, in the file `TableOrderV.js`, the following code:

```

  viewConfig: {
    plugins: {
      ptype: 'gridviewdragdrop',
      dragGroup: 'firstGridDDGroup',
      dropGroup: 'secondGridDDGroup'
    }
  },

```

Then, in the other grid we have to add the following code:

```

  viewConfig: {
    plugins: {
      ptype: 'gridviewdragdrop',
      dragGroup: 'secondGridDDGroup',
      dropGroup: 'firstGridDDGroup'
    }
  },

```

Navigate to our application and check that now we can drag and drop elements from the two grids.

In our case, what we want is to drag only from the first grid and drop in the second one. So we have to edit our configuration for this purpose.

Edit the configuration of the first grid adding the property `enableDrop` with the value to false:

```

  viewConfig: {
    plugins: {
      ptype: 'gridviewdragdrop',
      dragGroup: 'firstGridDDGroup',
      dropGroup: 'secondGridDDGroup',
      enableDrop: false
    }
  },

```

Edit the configuration of the second one adding the property `enableDrag` with value to false:

```

  viewConfig: {
    plugins: {
      ptype: 'gridviewdragdrop',
      dragGroup: 'secondGridDDGroup',
      dropGroup: 'firstGridDDGroup',
      enableDrag: false
    }
  },

```

Check the changes. Now, we can only drag from the first grid and drop to the second one.

Besides this, what we want is to drag from the first grid but keep the record in the grid as we can have the same item more than once. In order to do this we need to edit the configuration of the Order grid, the second grid. We have to add a listener for the drop in the configuration of the drag and drop:

```

  listeners: {
    drop: 'restoreMenu'
  }

```

When the drop is launched we are going to execute the function `restoreMenu`. In our `ViewControllerTableOrderVC.js` we have to define the function:

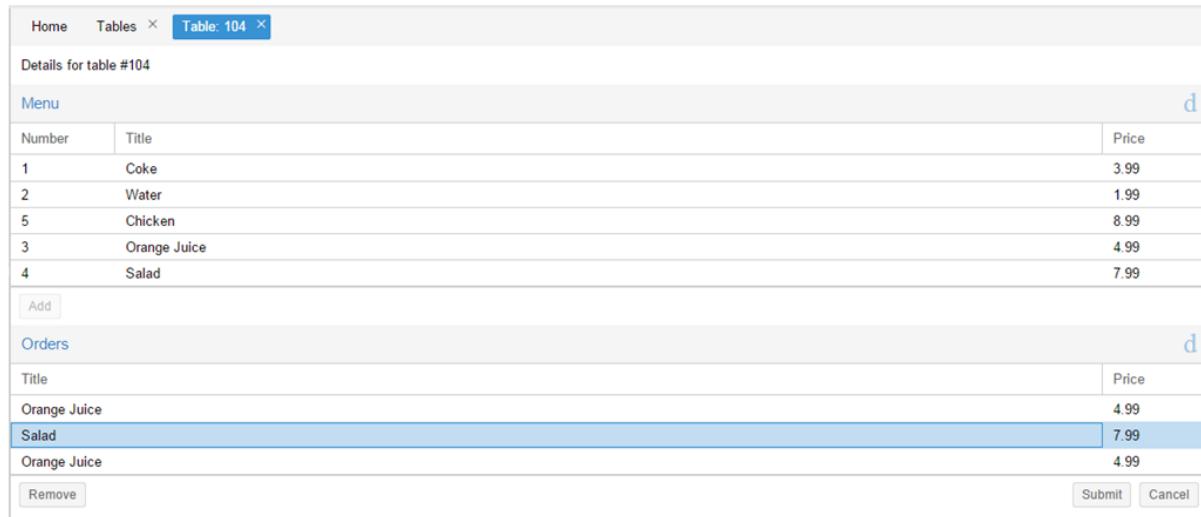
```

  restoreMenu: function(node, data, dropRec, dropPosition){
    var record = data.records[0].data;
    var vm = this.getViewModel();
    vm.get("menu").add(record);
  }

```

We are adding again the element that we have just dropped from the first grid.

Navigate to the application to check the results



Number	Title	Price
1	Coke	3.99
2	Water	1.99
5	Chicken	8.99
3	Orange Juice	4.99
4	Salad	7.99

Add

Title	Price
Orange Juice	4.99
Salad	7.99
Orange Juice	4.99

Remove

Submit Cancel

## 3.48 Extending the CRUD: Working with layouts

We are going to do a simple exercise to practice with **layouts**. Previously we have created a new view to order some products from a menu. We have two grids but they are showed one below the other. We are going to change the way they are displayed on the screen.

Therefore, our goal here is to show the grids as you can see below:

Home Tables Table: 101

Details for table #101

Menu		Orders	
Add	Edit	Delete	
Number	Title	Price	
2	Water	1.99	
4	Salad	7.99	
5	Chicken	8.99	
3	Orange Juice	4.99	
1	Coke	3.99	

Orders	
Title	Price
Orange Juice	4.99
Coke	3.99

In order to get that output we should make some changes in our code.

- Edit the **view** `TableOrderV.js` adding the layout 'vbox'

```
layout: {
    type: 'vbox',
    align: 'stretch'
},
```

and creating a new panel to englobe the two grids with the layout 'hbox' so they are going to be aligned horizontally:

```
{
    xtype: 'panel',
    layout: {
        type: 'hbox',
        align: 'stretch'
    },
    flex: 1,
    padding: 10,
    border: false,
    items: [
        this.gridMenu(),
        this.gridOrder()
    ]
}
```

This is the complete code for the view with the changes mentioned above:

```
Ext.define("demo.view.table.TableOrderV", {
    extend: "Ext.panel.Panel",
    alias: 'widget.tableorder',
    requires: [
        'demo.view.table.TableOrderVM',
        'demo.view.table.TableOrderVC'
    ],
    controller: "table-order-controller",
    viewModel: {
        type: "table-order-model"
    },
    items: [
        {
            xtype: 'panel',
            layout: {
                type: 'hbox',
                align: 'stretch'
            },
            flex: 1,
            padding: 10,
            border: false,
            items: [
                this.gridMenu(),
                this.gridOrder()
            ]
        }
    ]
});
```

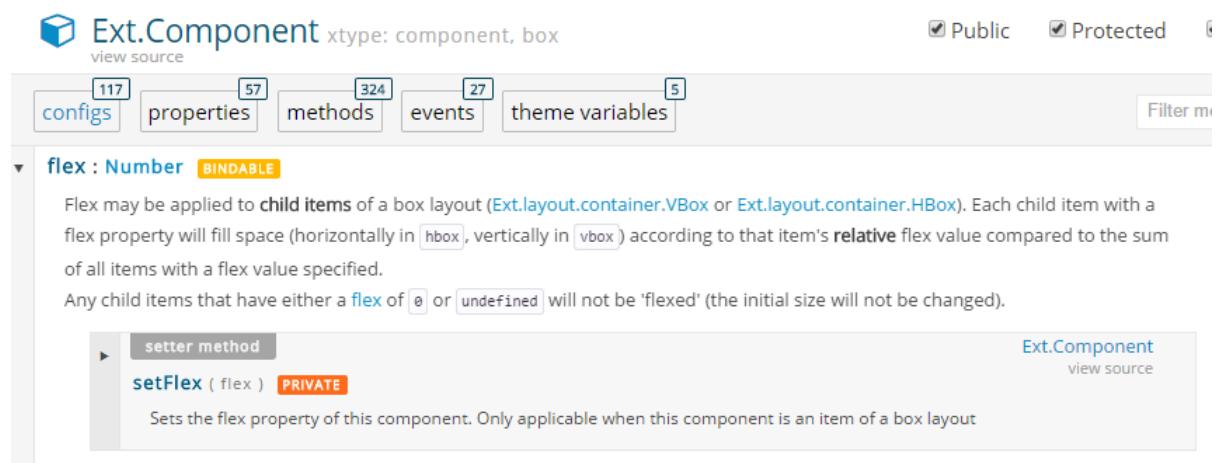
```
layout: {
    type: 'vbox',
    align: 'stretch'
},
closable: true,
initComponent: function() {
    Ext.apply(this, {
        items : [
            this.header(),
            {
                xtype: 'panel',
                layout: {
                    type: 'hbox',
                    align: 'stretch'
                },
                flex: 1,
                padding: 10,
                border: false,
                items: [
                    this.gridMenu(),
                    this.gridOrder()
                ]
            }
        ]
    });
    this.callParent(arguments);
},
header: function(){
    return {
        padding: 10,
        bind: {
            html: i18n.tableOrder.html + '{tableId}'
        },
        border: false
    }
},
gridMenu: function(){
    return {
        xtype: 'grid',
        title: 'Menu',
        flex: 1,
        margin: '0 10 0 0',
        reference: 'menugrid',
        allowDeselect: true,
        columns: [{
            text: i18n.tableOrder.grid.number,
            dataIndex: 'id'
        }, {
            text: i18n.tableOrder.grid.title,
            dataIndex: 'description',
            flex: 1
        }, {
            text: i18n.tableOrder.grid.price,
            dataIndex: 'price'
        }],
        bind: {
            store: '{menu}',
            selection: '{selectedMenuItem}'
        },
        viewConfig: {
            plugins: {
                ptype: 'gridviewdragdrop',
                dragGroup: 'firstGridDDGroup',
                dropGroup: 'secondGridDDGroup',
                enableDrop: false
            }
        }
    },
}
```

```
        tbar: [
            items: [
                {
                    text: i18n.tables.buttons.add,
                    handler: 'onAddMenuItemClick'
                },
                {
                    text: i18n.tables.buttons.edit,
                    bind: {
                        disabled: '{!selectedMenuItem}'
                    },
                    handler: 'onEditMenuItemClick'
                },
                {
                    text: i18n.tables.buttons.del,
                    bind: {
                        disabled: '{!selectedMenuItem}'
                    },
                    handler: 'onDeleteMenuItemClick'
                }
            ],
            bbar: [
                items: [
                    {
                        text: i18n.tableOrder.add,
                        bind: {
                            disabled: '{!selectedMenuItem}'
                        },
                        handler: 'addOrder'
                    }
                ]
            ],
            listeners: {
                beforeitemdblclick: 'onEditMenuItemDoubleClick'
            }
        }
    },
    gridOrder: function(){
        return {
            xtype: 'grid',
            title: 'Orders',
            flex: 1,
            margin: '0 0 0 10',
            reference: 'ordergrid',
            allowDeselect: true,
            columns: [
                {
                    dataIndex: 'id',
                    hidden: true
                },
                {
                    text: i18n.tableOrder.grid.title,
                    dataIndex: 'description',
                    flex: 1
                },
                {
                    text: i18n.tableOrder.grid.price,
                    dataIndex: 'price'
                }
            ],
            bind: {
                store: '{order}',
                selection: '{selectedOrderItem}'
            },
            viewConfig: {
                plugins: {
                    ptype: 'gridviewdragdrop',
                    dragGroup: 'secondGridDDGroup',
                    dropGroup: 'firstGridDDGroup',
                    enableDrag: false
                },
                listeners: {
                    drop: 'restoreMenu'
                }
            },
            bbar: [
                items: [
                    {
                        text: i18n.tableOrder.remove,

```

```
        bind: {
            disabled: '{!selectedOrderItem}'
        },
        handler: 'orderRemove'
    },
    '->', {
        text: i18n.tableOrder.submit,
        handler: 'tableOrderSubmit'
    }, {
        text: i18n.tableOrder.cancel,
        handler: 'tableOrderCancel'
    }
]
}
}
});
```

For more information about the configuration property `flex` check Sencha documentation:



**Ext.Component** xtype: component, box

Public  Protected

View source

117 configs 57 properties 324 methods 27 events 5 theme variables

**flex : Number** BINDABLE

Flex may be applied to child items of a box layout (`Ext.layout.container.VBox` or `Ext.layout.container.HBox`). Each child item with a flex property will fill space (horizontally in `hbox`, vertically in `vbox`) according to that item's **relative** flex value compared to the sum of all items with a flex value specified.

Any child items that have either a `flex` of `0` or `undefined` will not be 'flexed' (the initial size will not be changed).

setter method

**setFlex** ( flex ) PRIVATE

Sets the flex property of this component. Only applicable when this component is an item of a box layout

## 3.49 Extending the CRUD: Grid editable

In this section, we are going to develop a grid that allows the modification of its fields without having to open a new window.

### 3.49.1 Creating the grid

We are going to create a list of employees and edit the cells of the grid.

- **Step 1:** First of all, create a new component in the menu app/view/main/menu.js.

```
items: [{  
    xtype: 'toolbar',  
    items: [{  
        text: 'Tables', // i18n.main.menu.tables,  
        eventName: 'eventOpenTableList'  
    }, {  
        text: 'Employees', // i18n.main.menu.employees,  
        eventName: 'eventOpenEmployeeView'  
    }]  
},
```

- **Step 2:** The next step is to create the general **controller** that will listen to the event from the menu and will trigger it in order to create the new view.

Create the file app/controller/employee/EmployeesController.js:

```
Ext.define('demo.controller.employee.EmployeeController', {  
    extend: 'Ext.app.Controller',  
    /*view requires*/  
    requires:[  
        'demo.view.employee.i18n.Employee_en_EN',  
        'demo.view.employee.EmployeeV'  
    ],  
  
    /*Global events listeners definition*/  
    config: {  
        listen: {  
            global: {  
                eventOpenEmployeeView: 'onMenuOpenEmployee'  
            }  
        }  
    },  
  
    /*Create a view when a global event is fired*/  
  
    onMenuOpenEmployee: function(options) {  
        var employees = new demo.view.employee.EmployeeV(options);  
        Devon.App.openInContentPanel(employees);  
    }  
});
```

- **Step 3:** Add the controller in the file Application.js:

```
controllers: [  
    'demo.controller.main.MainController',  
    'demo.controller.page1.Page1Controller',  
    'demo.controller.table.TableController',  
    'demo.controller.employee.EmployeeController'  
],
```

- **Step 4:** Create the View Model.

Create the file app/view/employee/EmployeeVM.js:

```
Ext.define('demo.view.employee.EmployeeVM', {
  extend: 'Ext.app.ViewModel',
  alias: 'viewmodel.employees',

  data: {

  },

  stores: {
    employees: {
      fields: ['id','name','rol'],
      data:[
        {'id': 1, 'name':'Anthony Jonhson', 'rol':'waiter'},
        {'id': 2, 'name':'Allen Davis', 'rol':'barkeeper'},
        {'id': 3, 'name':'Robert Paul Juice', 'rol':'chief'},
        {'id': 4, 'name':'Michael Thompson', 'rol':'barman'},
        {'id': 5, 'name':'John Stone', 'rol':'cook'}
      ]
    }
  }
});
```

- **Step 5:** Then, create the view for the list of employees

Create the file app/view/employee/EmployeeV.js:

```
Ext.define("demo.view.employee.EmployeeV", {
  extend: "Ext.panel.Panel",
  alias: 'widget.employees',
  requires: [
    'Ext.grid.Panel',
    'demo.view.employee.EmployeeVM'
  ],

  viewModel: {
    type: "employees"
  },
  layout: {
    type: 'vbox',
    align: 'stretch'
  },
  title: i18n.employees.title,
  closable: true,
  initComponent: function() {
    Ext.apply(this, {
      items : [this.grid()]
    });
    this.callParent(arguments);
  },

  grid: function(){
    return {
      xtype: 'grid',
      reference: 'grid',
      title: i18n.employees.title,
      flex:1,
      margin: 10,
      allowDeselect: true,
      columns: [{
        text: i18n.employees.grid.name,
        dataIndex: 'name',
        flex: 1
      }, {
        text: i18n.employees.grid.rol,
        dataIndex: 'rol'
      }],
      bind: {
        store: '{employees}'
      }
    }
  }
});
```

} ) ; } }

- **Step 6:** Finally, we have to create the file with the internationalized texts.

Create the file app/view/employee/i18n/Employee\_en\_EN.js:

```
Ext.define('demo.view.employee.i18n.Employee_en_EN', {
    extend: 'Devon.I18nBundle',
    singleton: true,
    i18n: {
        employees: {
            title: 'Employees',
            grid: {
                name: 'Name',
                rol: 'Rol'
            }
        }
    }
});
```

Now, check the output and see that we have created a list of employees:



The screenshot shows a web application interface for a restaurant management system. At the top, there is a logo for "devonfw" and a "demo" label. On the right, there are language selection ("English") and log off buttons. Below the header, there are navigation buttons for "Tables" and "Employees". The main content area shows a breadcrumb navigation path: "page1" → "My new page" → "Employees". On the left, a sidebar titled "Restaurant" contains a "Tables" section with three items: "Free (4)", "Reserved (0)", and "Occupied (1)". The main content area is titled "Employees" and displays a table with the following data:

Name	Role
Anthony Jonhson	waiter
Allen Davis	barkeeper
Robert Paul Juice	chief
Michael Thompson	barman
John Stone	cook

### 3.49.2 Cellediting plugin

Once the base of our view is defined with a list of employees, we are going to add some functionality to have an editable grid. First of all, we are going to add a plugin to our grid in order to edit cells.

Edit the file app/view/employee/EmployeeV.js:

```
grid: function(){
  return {
    xtype: 'grid',
    reference: 'grid',
    title: i18n.employees.title,
    flex:1,
    margin: 10,
    allowDeselect: true,
    plugins: [
      ptype: 'cellediting',
    ],
  }
}
```

```

        clicksToEdit: 2
    },
    columns: [
        ...

```

Now, we can configure column by column the editor type that we want for each of them. By default, if any type of editor has been specified (textfield, numberfield, datefield, etc.) the type will be `textfield`.

Let's add a `textfield` editor to the `Name` column. Edit the file `app/view/employee/EmployeeV.js`:

```

columns: [
    {
        text: i18n.employees.grid.name,
        dataIndex: 'name',
        flex: 1,
        editor: {
            allowBlank: false
        }
    },
    {
        text: i18n.employees.grid.rol,
        dataIndex: 'rol'
    }
]

```

Check that we can edit the column `Name`:

Name	Rol
Anthony Jonhson	waiter
Allen Davis	barkeeper
Robert Paul Juice	chief
Michael Thompson	barman
John Stone	cook

### Combobox as an editing cell

Modify the column `Rol` to be editable as a combobox component.

Add a new store in the file `app/view/employee/EmployeeVM.js`:

```

stores: {
    employees: {
        fields: ['id', 'name', 'rol'],
        data:[
            {'id': 1, 'name':'Anthony Jonhson', 'rol':'waiter'},
            {'id': 2, 'name':'Allen Davis', 'rol':'barkeeper'},
            {'id': 3, 'name':'Robert Paul Juice', 'rol':'chief'},
            {'id': 4, 'name':'Michael Thompson', 'rol':'barman'},
            {'id': 5, 'name':'John Stone', 'rol':'cook'}
        ]
    },
    roles: {
        fields: ['rol'],
        data:[

```

```

        {'rol':'waiter'},
        {'rol':'barkeeper'},
        {'rol':'chief'},
        {'rol':'barman'},
        {'rol':'cook'}
    ]
}
}

```

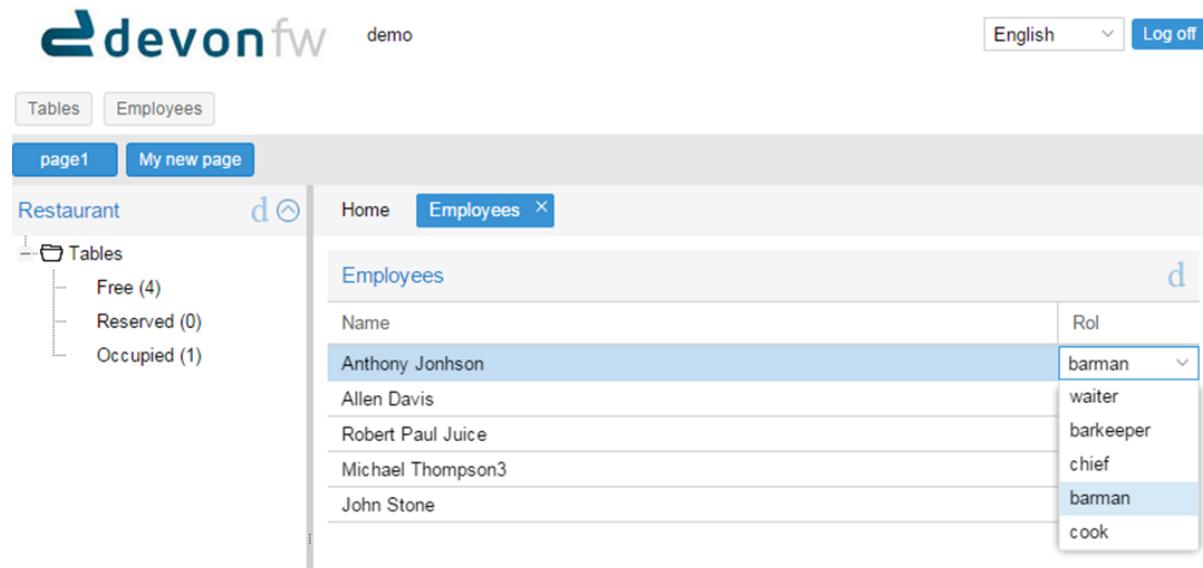
Modify the column Rol in app/view/employee/EmployeeV.js:

```

columns: [
  {
    text: i18n.employees.grid.name,
    dataIndex: 'name',
    flex: 1,
    editor: {
      allowBlank: false
    }
  },
  {
    text: i18n.employees.grid.rol,
    dataIndex: 'rol',
    editor: {
      xtype: 'combobox',
      bind: {
        store: '{roles}'
      },
      displayField: 'rol',
      valueField: 'rol',
      editable: false,
      queryMode: 'local',
      forceSelection: true,
      triggerAction: 'all',
      allowBlank: false
    }
  },
]

```

Check the output in the screen with the changes we have made:



The screenshot shows the 'Employees' grid in the 'Employees' view. The 'Rol' column is a dropdown combobox. The dropdown menu is open, showing the following options: 'barman', 'waiter', 'barkeeper', 'chief', 'barman', and 'cook'. The option 'barman' is highlighted with a blue background. The grid data is as follows:

Name	Rol
Anthony Jonhson	barman
Allen Davis	waiter
Robert Paul Juice	barkeeper
Michael Thompson3	chief
John Stone	barman
	cook

## Dirty

Note that, after editing a cell, it shows a red mark. It indicates that the cell is **dirty**, it means that the changes have not been confirmed. For that purpose, after editing the grid we should send the data to the server to confirm it or just commit the change in the view.

The first option is the most usual. However, for this example, we are going to take the second option.

Modify the file `app/view/employee/EmployeeV.js` adding a listener to commit the change in the grid:

```
grid: function(){
  return {
    xtype: 'grid',
    reference: 'grid',
    ...

    bind: {
      store: '{employees}'
    },
    listeners: {
      edit: function(editor, e) {
        e.record.commit();
      }
    }
  }
}
```

We could have created a **ViewController** to manage the listener but we have decided to simplify the example and show another way to treat the event.

Navigate to the application and check that there is not red mark anymore after editing a cell.

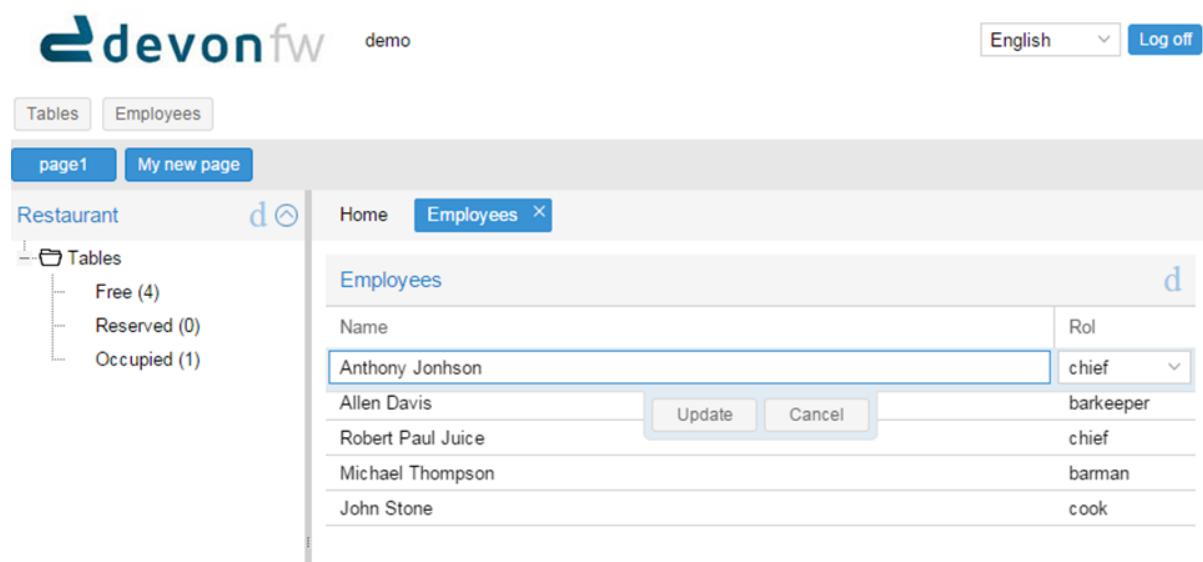
### 3.49.3 Rowediting plugin

In this example we allow to edit the grid cell by cell, but if we want to edit the whole row at a time, we just have to change the type of plugin used.

Modify the type of plugin in the view `app/view/employee/EmployeeV.js`:

```
plugins: {
  ptype: 'rowediting',
  clicksToEdit: 2
},
```

Navigate to the application and check the new behaviour when editing the grid:



Name	Rol
Anthony Jonhson	chief
Allen Davis	barkeeper
Robert Paul Juice	chief
Michael Thompson	barman
John Stone	cook

### 3.49.4 Add empty records

Now, we want to add new records to the store.

We are going to add a button Add in the bbar property of the grid and create an empty record for our store. As the grid is editable we would fill the data and have a new record for our employee list.

- **Step 1:** The first step is to create a new button in the grid:

```
bbar: {
    items: [
        {
            text: i18n.employees.add,
            handler: 'addEmployee'
        }
    ]
},
```

- **Step 2:** Create the **viewController** EmployeeVC.js in app/view/employee/ folder:

```
Ext.define('demo.view.employee.EmployeeVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.employee-controller',

    addEmployee: function() {
        var vm = this.getViewModel();

        vm.get("employees").add({
            id: null,
            name: null,
            rol: null
        });
    }
});
```

- **Step 3:** Add the reference in the file Employee\_en\_EN.js for the button:

```
Ext.define('demo.view.employee.i18n.Employee_en_EN', {
    extend: 'Devon.I18nBundle',
    singleton: true,
    i18n: {
        employees: {
            title: 'Employees',
            grid: {
                name: 'Name',
                rol: 'Rol'
            },
            add: 'Add'
        }
    }
});
```

- **Step 4:** Finally, add the reference of the **ViewController** in the view:

```
requires: [
    'Ext.grid.Panel',
    'demo.view.employee.EmployeeVM',
    'demo.view.employee.EmployeeVC'
],  
  
viewModel: {
    type: "employees"
},  
controller: 'employee-controller',
```

Navigate to the application and check that when we click on the button Add a new empty record is created in the grid ready to be edited.

## 3.50 Extending the CRUD: Custom plugin

We have seen how to use the plugins that are provided by default by Sencha as the cellediting or rowediting plugins.

Now, we are going to see a simple example of how to create our own plugin. For this purpose we want to create a plugin to manage the permissions of specific functionalities depending on the roles of the user logged in the application. Therefore, depending on the roles the buttons will be hidden or shown.

We just need to create a new file, for instance, `demo/plugins/SecureAccess.js` where we will manage :

```
Ext.define('demo.plugins.SecureAccess', {
    extend: 'Ext.plugin.Abstract',
    alias: 'plugin.secureaccess',

    requires: [
        'Ext.button.Button',
        'Ext.toolbar.Toolbar'
    ],

    init : function(component) {
        if(Devon.Security && Devon.Security.currentUser && Devon.Security.currentUser.roles){
            var userRoles = Devon.Security.currentUser.roles;

            var roleInList = function(componentRoles){
                for(var rol in componentRoles){
                    if(userRoles[componentRoles[rol]]){
                        return true;
                    }
                }
            }

            return false;
        };

        if (component.items && component.items.items) {
            var items = component.items.items;

            for(var pos in items){
                var item = items[pos];

                if(item.roles && !roleInList(item.roles)){
                    item.hide();
                }
            }
        }
    }
});
```

Now, we have to add the plugin recently created to use the functionality. For example, we can apply this behaviour for allowing or disallowing add, edit and delete tables in our application.

In the file `TableListV.js` add the reference to the plugin the in requires property:

```
requires: [
    'Ext.grid.Panel',
    'Devon.grid.plugin.Pagination',
    'demo.plugins.SecureAccess',
    'demo.view.table.TableListVM',
    'demo.view.table.TableListVC'
],
```

Then, in the toolbar of the grid we add the plugin and the roles for the buttons:

```
tbar: {
    plugins:['secureaccess'],
    items: [{{
        text: i18n.tables.buttons.add,
        handler: 'onAddClick',
        roles:[ 'WAITER' ]
    }}, {
        text: i18n.tables.buttons.edit,
        bind: {
            disabled: '{!selectedItem}'
        },
        roles:[ 'WAITER' ],
        handler: 'onEditClick'
    }, {
        text: i18n.tables.buttons.del,
        roles:[ 'CHIEF' ],
        bind: {
            disabled: '{!selectedItem}'
        },
        handler: 'onDeleteClick'
    }
    ...
}
```

Navigate to the application and check how the buttons appear or disappear depending on the user and the roles. Try with `waiter` and `chief`. The user `waiter` should be able to add and edit tables. The user `chief` should be able to do all the functionalities.

## 3.51 The class system

Ext JS provides a number of functions that make it simple to create and work with classes. The following are the classes in the Ext JS 6 class system:

- Ext
- Ext.Base
- Ext.Class
- Ext.ClassManager
- Ext.Loader

### 3.51.1 Ext

Ext is a global singleton object that encapsulates all classes, singletons, and utility methods in the Sencha library. Many commonly used utility functions are defined in Ext. It also provides shortcuts to frequently used methods in other classes.

## 3.52 Components

An Ext JS application's UI is made up of one or many widgets called Components. All Components are subclasses of the Ext.Component class which allows them to participate in automated lifecycle management including instantiation, rendering, sizing and positioning, and destruction. Ext JS provides a wide range of useful Components out of the box, and any Component can easily be extended to create a customized Component.

To know more about Components visit [Sencha](#) website.

### 3.52.1 Creating Custom Components

#### Composition or Extension

When creating a new UI class, the decision must be made whether that class should own an instance of a Component, or to extend that Component.

It is recommended to extend the nearest base class to the functionality required. This is because of the automated lifecycle management Ext JS provides which includes automated rendering when needed, automatic sizing and positioning of Components when managed by an appropriate layout manager, and automated destruction on removal from a Container.

It is easier to write a new class which is a Component and can take its place in the Component hierarchy rather than a new class which has an Ext JS Component, and then has to render and manage it from outside.

#### Subclassing

The Class System makes it easy to extend any part of the Ext JS framework.

Ext.Base is the building block of all Ext JS classes, and the prototype and static members of this class are inherited by all other classes.

While you can certainly begin adding functionality at the lowest possible level via Ext.Base, in many cases developers want to start a bit higher in the inheritance chain.

The following example creates a subclass of Ext.Component:

```
Ext.define('My.custom.Component', {
    extend: 'Ext.Component',

    newMethod : function() {
        //...
    }
});
```

This example creates a new class, My.custom.Component, which inherits all of the functionality (methods, properties, etc.) of Ext.Component in addition to any new methods or properties defined.

### Template Methods

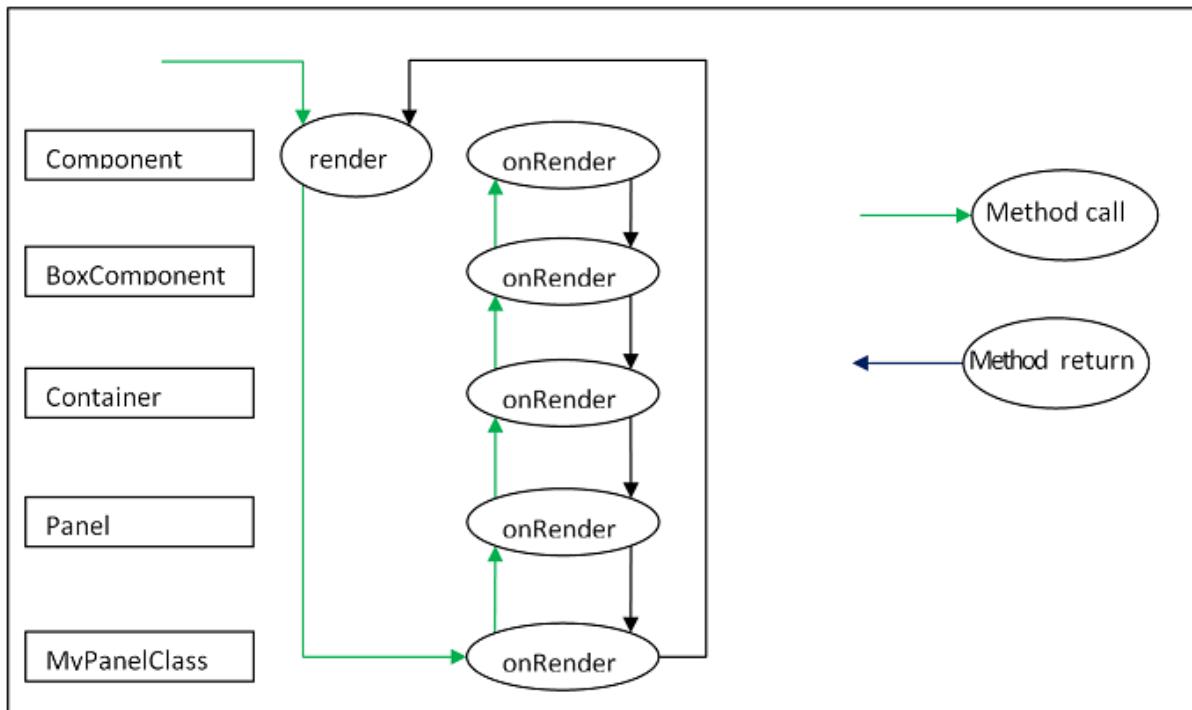
Ext JS uses the Template method pattern to delegate to subclasses, behavior which is specific only to that subclass.

This means each class in the inheritance chain may "contribute" an extra piece of logic to certain phases in the Component's lifecycle. Each class implements its own special behavior while allowing the other classes in the inheritance chain to continue to contribute their own logic.

An example is the render function. render is a method defined in Component. It is responsible for initiating the rendering phase of the Component lifecycle. render must not be overridden, but it calls onRender during processing to allow the subclass implementor to add an onRender method to perform class-specific processing. Every onRender method must call its superclass' onRender method before "contributing" its extra logic.

The diagram below illustrates the functioning of the onRender template method.

The render method is called (This is done by a Container's layout manager). This method may not be overridden and is implemented by the Ext base class. It calls this.onRender which is the implementation within the current subclass (if implemented). This calls the superclass version which calls its superclass version etc. Eventually, each class has contributed its functionality, and control returns to the render function.



Here is an example of a Component subclass that implements the onRender method:

```
Ext.define('My.custom.Component', {
    extend: 'Ext.Component',
    onRender: function() {
        this.callParent(arguments); // call the superclass onRender method

        // perform additional rendering tasks here.
    }
});
```

It is important to note that many of the template methods also have a corresponding event. For example the render event is fired after the Component is rendered. When subclassing, however, it is essential to use template methods to perform class logic at important phases in the lifecycle and not events. Events may be programmatically suspended, or may be stopped by a handler.

Below are the template methods that can be implemented by subclasses of Component:

- **initComponent** This method is invoked by the constructor. It is used to initialize data, set up configurations, and attach event handlers.
- **beforeShow** This method is invoked before the Component is shown.
- **onShow** Allows addition of behavior to the show operation. After calling the superclass's onShow, the Component will be visible.
- **afterShow** This method is invoked after the Component is shown.
- **onShowComplete** This method is invoked after the afterShow method is complete
- **onHide** Allows addition of behavior to the hide operation. After calling the superclass's onHide, the Component will be hidden.
- **afterHide** This method is invoked after the Component has been hidden

- `onRender` Allows addition of behavior to the rendering phase.
- `afterRender` Allows addition of behavior after rendering is complete. At this stage the Component's Element will have been styled according to the configuration, will have had any configured CSS class names added, and will be in the configured visibility and the configured enable state.
- `onEnable` Allows addition of behavior to the enable operation. After calling the superclass's `onEnable`, the Component will be enabled.
- `onDisable` Allows addition of behavior to the disable operation. After calling the superclass's `onDisable`, the Component will be disabled.
- `onAdded` Allows addition of behavior when a Component is added to a Container. At this stage, the Component is in the parent Container's collection of child items. After calling the superclass's `onAdded`, the `ownerCt` reference will be present, and if configured with a ref, the `refOwner` will be set.
- `onRemoved` Allows addition of behavior when a Component is removed from its parent Container. At this stage, the Component has been removed from its parent Container's collection of child items, but has not been destroyed (It will be destroyed if the parent Container's `autoDestroy` is true, or if the `remove` call was passed a truthy second parameter). After calling the superclass's `onRemoved`, the `ownerCt` and the `refOwner` will not be present.
- `onResize` Allows addition of behavior to the resize operation.
- `onPosition` Allows addition of behavior to the position operation.
- `onDestroy` Allows addition of behavior to the destroy operation. After calling the superclass's `onDestroy`, the Component will be destroyed.
- `beforeDestroy` This method is invoked before the Component is destroyed.
- `afterSetPosition` This method is invoked after the Components position has been set.
- `afterComponentLayout` This method is invoked after the Component is laid out.
- `beforeComponentLayout` This method is invoked before the Component is laid out.

### Which Class To Extend

Choosing the best class to extend is mainly a matter of efficiency, and which capabilities the base class must provide. There has been a tendency to always extend `Ext.panel.Panel` whenever any set of UI Components needs to be rendered and managed.

The Panel class has many capabilities:

- Border
- Header
- Header tools
- Footer
- Footer buttons

- Top toolbar
- Bottom toolbar
- Containing and managing child Components

If these are not needed, then using a Panel is a waste of resources.

## Component

If the required UI Component does not need to contain any other Components, that is, if it just to encapsulate some form of HTML which performs the requirements, then extending Ext.Component is appropriate. For example, the following class is a Component that wraps an HTML image element, and allows setting and getting of the image's src attribute. It also fires a load event when the image is loaded:

```
Ext.define('Ext.ux.Image', {
    extend: 'Ext.Component', // subclass Ext.Component
    alias: 'widget.managedimage', // this component will have an xtype of 'managedimage'

    autoEl: {
        tag: 'img',
        src: Ext.BLANK_IMAGE_URL,
        cls: 'my-managed-image'
    },

    // Add custom processing to the onRender phase.
    // Add a 'load' listener to the element.
    onRender: function() {
        this.autoEl = Ext.apply({}, this.initialConfig, this.autoEl);
        this.callParent(arguments);
        this.el.on('load', this.onLoad, this);
    },

    onLoad: function() {
        this.fireEvent('load', this);
    },

    setSrc: function(src) {
        if (this.rendered) {
            this.el.dom.src = src;
        } else {
            this.src = src;
        }
    },

    getSrc: function(src) {
        return this.el.dom.src || this.src;
    }
});
```

Usage:

```
var image = Ext.create('Ext.ux.Image');

Ext.create('Ext.panel.Panel', {
    title: 'Image Panel',
    height: 200,
    renderTo: Ext.getBody(),
    items: [ image ]
});

image.on('load', function() {
    console.log('image loaded: ', image.getSrc());
});

image.setSrc('http://www.sencha.com/img/sencha-large.png');
```

This example is for demonstration purposes only - the Ext.Img class should be used for managing images in a real world application.

## Container

If the required UI Component is to contain other Components, but does not need any of the previously mentioned additional capabilities of a Panel, then Ext.container.Container is the appropriate class to extend. At the Container level, it is important to remember which Ext.layout.container.Container is to be used to render and manage child Components.

Containers have the following additional template methods:

- **onBeforeAdd** This method is invoked before adding a new child Component. It is passed the new Component, and may be used to modify the Component, or prepare the Container in some way. Returning false aborts the add operation.
- **onAdd** This method is invoked after a new Component has been added. It is passed the Component which has been added. This method may be used to update any internal structure which may depend upon the state of the child items.
- **onRemove** This method is invoked after a new Component has been removed. It is passed the Component which has been removed. This method may be used to update any internal structure which may depend upon the state of the child items.
- **beforeLayout** This method is invoked before the Container has laid out (and rendered if necessary) its child Components.
- **afterLayout** This method is invoked after the Container has laid out (and rendered if necessary) its child Components.

## Panel

If the required UI Component must have a header, footer, or toolbars, then Ext.panel.Panel is the appropriate class to extend.

Important: A Panel is a Container. It is important to remember which Layout is to be used to render and manage child Components.

Classes which extend Ext.panel.Panel are usually highly application-specific and are generally used to aggregate other UI Components (Usually Containers, or form Fields) in a configured layout, and provide means to operate on the contained Components by means of controls in the tbar and the bbar.

Panels have the following additional template methods:

- **afterCollapse** This method is invoked after the Panel is Collapsed.
- **afterExpand** This method is invoked after the Panel is expanded
- **onDockedAdd** This method is invoked after a docked item is added to the Panel
- **onDockedRemove** This method is invoked after a docked item is removed from the Panel

## 3.53 Rest endpoints

Devon architecture for web applications is based on the standard JAX-RS for communication between client and server. The recommended use of REST is of a more "practical" approach and it is not oriented to **pure** RESTful (also known as hypermedia or [HATEOAS](#)) thus simplifying the programming both on the client and server side. (Please consult the platform guide for more information).

For easing the communication from Javascript code to the back-end, Devon provides helpers to define Rest endpoints as Javascript objects with methods to do a GET/POST/PUT/DELETE operations.

These Rest endpoints are usually created on the **Controllers**, so they get instanciated at application launch and then can be used within other **Controller** and **ViewController** instances.

On the sample application the **Rest endpoints** used for all the table related operations are created on the global Table Controller:

### Sample.controller.table.TableController.js.

```
init: function() {
  Devon.Ajax.define({
    'tablemanagement.table': {
      url: 'tablemanagement/v1/table/{id}'
    },
    'salesmanagement.order': {
      url: 'salesmanagement/v1/order'
    }
  });
},
```

These helper objects can be used in other controllers by calling get/post/put/delete methods following a similar syntax to the Ext.Ajax.request function.

- `Devon.rest.tablemanagement.table.get`
- `Devon.rest.tablemanagement.table.put`
- `Devon.rest.tablemanagement.table.post`
- `Devon.rest.tablemanagement.table.delete`

```
getTable: function(id) {
  Devon.rest.tablemanagement.table.get({
    scope: this,
    uriParams: {
      id: id
    },
    success: function(data, response, options){
      // work with response data
    }
  });
},
```

Here a GET request is made by the client to the following url:

```
[BASE_URL] /tablemanagement/v1/table/123
```

The `Devon.Ajax.define` objects wrap `Ext.Ajax.request` methods with some added functionality:

- \* URL template resolution
- \* Automatic JSON response decode
- \* Default failure function

### 3.53.1 URL template resolution

The `url` parameter used to define a route is treated as an `Ext.XTemplate` that is resolved with the `uriParams` property when the method is called.

This way an URL like `/someurl/{id}` is converted to `/someurl/123` when called with `uriParams: {id:123}`. If there is no `id` property on the parameters it becomes blank string so what you get is `/someurl/`.

This is useful because allows us to reuse the same URL for both GET and POST requests. Remember that a GET request usually specifies the entity id on the path.

For example, to create or update an entity we make a post request:

```
Devon.rest.tablemanagement.table.post({
  scope: this,
  jsonData: {
    id: 123,
    name: "John",
    rol: "admin"
  }
});
```

which will lead to the following request to the server:

```
HTTP POST /tablemanagement/v1/table
Content-Type: application/json

{ "id" : 123, "name": "john", "rol": "admin" }
```

### 3.53.2 Query string

Any part of the url is subject to url substitution. It is therefore possible to add a so-called query string just by adding this, in parametrized format, to the url template. This way an URL template like

```
[BASE_URL]/someurl/{id}?format={docformat}
```

is converted to the url

```
[BASE_URL]/someurl/123?format=pdf
```

when called with `uriParams`

```
{id:123, docformat: "pdf"}
```

### 3.53.3 Alternative PUT format

An usual case for PUT operations is to address an individual resource on the URL by using its ID and sending the data on the body payload.

```
HTTP PUT /someurl/123
Content-Type: application/json

{
  "name" : "john",
  "rol" : "admin"
}
```

This can be easily accomplished by using `uriParams` and `jsonData` when calling the function.

```
Devon.rest.tablemanagement.table.put({
  scope: this,
  uriParams : { id : 123} ,
  jsonData: {
    name: "John",
    rol: "admin"
  }
});
```

### 3.53.4 Automatic JSON response decode

On the original `Ext.Ajax.request` function, the `success` callback property function takes 2 parameters ( `response` and `opts`). The first thing to do during this callback is usually decoding the `response.responseText` property of the `response` object from JSON to a Javascript object.

As JSON is assumed as the default communication between client and server, Devon helper methods already do this and introduce the decoded Javascript object as first parameter to the `success` function.

This makes easier also to delegate in a more clean way to other functions defined on the same **controller** and avoid the so called "callback pyramid of doom"

for example:

```
getTable: function(id) {
  Devon.rest.tablemanagement.table.get({
    scope: this, ❶
    uriParams: { id: 123 },
    success: this.loadOrder ❷
  });
},
loadOrder: function(table, response, opts) { ❸
  // table is already a Javascript object
}
```

- ❶ scope for the success function
- ❷ delegate to `loadOrder` function on the same class
- ❸ You usually doesn't need to declare and use `response` and `opts` in the callback

### 3.53.5 Base Url configuration

All the rest endpoints of the application are configured to use the same base url configured in the `Config.js` settings file with the **server** property:

```
window.Config={
  ...
  server : '/devonfw-sample-server/services/rest/',
  ...
};
```

## 3.54 Sencha data package

Sencha provides a powerful data package with **models**, **stores** and **associations**. But this power comes with added complexity added that usually doesn't justify the price.

For example, the Sencha way of loading an entity is by providing a `proxy` to a `model` and calling the `load` method. This fires an asynchronous operation that when resolved, completes with the `model` being populated with the information from the server.

Models and stores are useful for some controls of the Sencha framework like combo, list, grid or form.

```
// Table model is configured with a rest proxy

new Sample.model.table.Table().load(123, {
  success: function(){
    // model is loaded, proceed with logic
  }
});
```

In Devon we usually recommend explicitly calling a "**rest endpoint**" and create the `model` in the callback (if necessary, maybe you can work with the returned data without really creating a `model`)

```
Devon.rest.tablemanagement.table.get({
  params : { id : 123 },
  success: function(data){
    // work with data or... create a model
    var model = new Sample.model.table.Table(data);
  }
});
```

Things get a bit complicated when dealing with associations between nested models. Reading a nested structure into a model is an easy task but "writing" back this nested structure is not. Usually it is better to work with good plain old Javascript objects and use a "rest endpoint" to do the communication with the server.

### 3.54.1 Devon.restproxy

When defining rest endpoints through the `Devon.Ajax.define` method, it also defines a Rest proxy with the same type as the key of the endpoint. For example for this endpoint:

```
Devon.Ajax.define({
  'tablemanagement.table' : {
    url: 'tablemanagement/v1/table/{id}'
  }
});
```

An Ajax proxy is created of type `Devon.restproxy.tablemanagement.table` with an alias of `proxy.tablemanagement.table` that can be used by combos or grids. For example, when loading a combo through a `viewModel` object on the Sample application:

```
Ext.define('Sample.view.table.TableEditVM', {
  extend: 'Ext.app.ViewModel',
  ...
  stores: {
    offers: {
      model: 'Sample.model.Offer',
      proxy : {
        type: 'tablemanagement.offer'
      },
      autoLoad: true
    }
  }
});
```

```
    }
}
});
```

This proxy follows the convention in Devon to use POST for performing read requests that contain additional filters or pagination and sorting information.

### 3.54.2 Pagination

When we have a lot of results for a 'READ' operation, we can not show all the results at the same time due to performance issues. The solution is to paginate the results. In Sencha, we can define a [paginated data store](#) setting `pageSize` with a number greater than 0. The user can navigate between data pages with the pagination toolbar of a [Sencha grid component](#).

So, to use pagination you only have to configure three components in Devon4Sencha:

- **REST endpoint:** Activating `pagination` flag in the controller REST endpoint definition. For example:

```
init: function() {
  Devon.Ajax.define({
    'tablemanagement.search': {
      url: 'tablemanagement/v1/table/search',
      pagination : true
    },
    ...
  });
}
```

- **Store:** Configure a store class with the paginated rest endpoint proxy and a value for `pageSize` greater than 0:

```
stores: {
  tables: {
    model: 'Sample.model.table.Table',
    pageSize: 3,
    proxy: {
      type : 'tablemanagement.search'
    },
    autoLoad:true
  }
}
```

- **Grid.** Add pagination plugin. This plugin adds a pagination toolbar to a grid and binds the grid store to the pagination toolbar. For example:

```
{
  xtype: 'grid',
  bind: {
    store: '{tables}'
  },
  plugins:[ 'pagination'],
  ...
}
```

### 3.54.3 Sorting

By default, [Sencha grid component](#) allows the user to sort the visible data from the grid by clicking on the header columns. We can activate the store property `remoteSort` if we want to implement sorting in the back-end:

```
stores: {
  tables: {
    model: 'Sample.model.table.Table',
    pageSize: 3,
    proxy: {
      type : 'tablemanagement.search'
    },
    remoteSort : true,
    autoLoad:true
  }
}
```

The `remoteSort` property, makes Sencha framework to send the additional params on the body request object to the back-end:

```
sort: "[{"property":"state","direction":"ASC"}]"
```

Where `property` has the name of the model `property` for the column selected and `direction` has the value "ASC" (ascending) or "DESC" (descending).

## 3.55 Ext.window.Window

Windows are a Panel subclass designed to float above all other components in an application. Windows are a great way to present a popup-style context view to the end user. Used judiciously, windows can be a great way to present a lot of information without intruding on the other views occupying the screen real estate.

By default, windows will be rendered to the document body (document.body), though they may also be rendered to an HTML element using the renderTo config option. You can also add windows as child components of a container. Windows added as a child item will not participate in the component's layout, but will be hidden when the parent container is hidden and destroyed when the parent container is destroyed. The window will also have access to the parent container's view model data and view controller logic.

Note: For more information on view controller event handling and data binding via the view model see the view controller and view model guides.

### 3.55.1 Basic Window

Windows, like other floating components, are initially rendered as hidden and must be shown programmatically using the show method.

```
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    buttons: [{
        text: 'Login'
    }]
});

win.show(); // once created, call show to display the window
```

When showing a window you can have it appear as though it's emerging from an existin component or DOM element. Simply configure the window with animateTarget and set the element or component you want the window to pop up from.

```
var btn = Ext.create({
    xtype: 'button',
    renderTo: Ext.getBody(),
    text: 'Show User Login',
    handler: function () {
        win.show();
    }
});

var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    closeAction: 'hide', // hides instead of destroys the window when closed
    animateTarget: btn,
    buttons: [{
        text: 'Login'
    }]
});
```

If you want to show a window as soon as it's created you can use the autoShow config option.

```
Ext.create({
```

```
xtype: 'window',
title: 'User Login',
height: 200,
width: 200,
autoShow: true, // shows the window as soon as its created
buttons: [{
    text: 'Login'
}]);
});
```

Often times when you show a window you'll want to restrict the user from interacting with the components residing beneath the window. This is easily done with the modal config option. Setting modal: true creates a semi-transparent overlay between the window and the application components below. The modal overlay intercepts all user interactions and redirects the user back to the context window.

```
var btn = Ext.create({
    xtype: 'button',
    renderTo: Ext.getBody(),
    text: 'Inaccessible button'
});
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    closable: false,
    autoShow: true,
    modal: true, // prevents users from interacting with components below
    buttons: [{
        text: 'Login'
    }]
});
```

The window will be centered within its parent container / element when shown. You can set the initial x / y position using the x and y config options. The x and y offsets are relative to the owning container / element versus the browser viewport.

```
// to demonstrate the ability to position the window relative to its
// owning container we'll create a container with two child containers
// the second of which will host our window
Ext.create({
    xtype: 'container',
    plugins: 'viewport',
    layout: {
        type: 'hbox',
        align: 'stretch'
    },
    items: [{
        xtype: 'container',
        flex: 1
    }, {
        xtype: 'container',
        flex: 1,
        items: [{
            xtype: 'window',
            title: 'User Login',
            height: 200,
            width: 200,
            x: 20, // offset the window 20px from the parent's left edge
            y: 20, // and 20px from the parent's top edge
            autoShow: true,
            buttons: [{
                text: 'Login'
            }]
        }]
    }]
});
```

Windows come with a convenient config for focusing a component once the window is shown. This comes in particularly handy when the window contains form elements. A common use case will be to focus the first input field when the window is shown.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    width: 240,
    autoShow: true,
    bodyPadding: 20,
    defaultFocus: 'textfield', // focuses on the first child textfield
    defaultType: 'textfield',
    defaults: {
        anchor: '100%'
    },
    layout: 'anchor',
    items: [
        {
            emptyText: 'login'
        },
        {
            emptyText: 'password'
        },
        {
            text: 'Login'
        }
    ]
});
```

Windows are draggable by default by clicking and dragging the window header. The window itself is not dragged, but rather a proxy representation of itself that indicates where the window will be dropped once the drag operation is complete. The window will drag outside of the owning container / element unless constrained. You have a couple of options available for preventing the window from being dragged outside its owning container. First, you can configure the window with constrain: true to prevent the window header or body from dragging outside the owner element's boundaries.

Note: You can prevent the window from being draggable by setting the draggable config option to false.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    constrain: true, // dragging is constrained within the owning container
    buttons: [
        {
            text: 'Login'
        }
    ]
});
```

Alternatively, you can specify that only the window's header is to be constrained. By using constrainHeader: true the window's header cannot be dragged outside of the owning container.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    // the window header cannot be dragged beyond the owning container
    constrainHeader: true,
    buttons: [
        {
            text: 'Login'
        }
    ]
});
```

Using the liveDrag config you can drag the window component itself rather than a proxy of the window.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    liveDrag: true,
    buttons: [{
        text: 'Login'
    }]
});
```

Windows are resizable by default allowing you to resize the window by dragging a corner or any side in order to change the dimensions of the window. You can prohibit resizing by setting resizable: false. Or, if the use case dictates, you can pass a config object for the underlying resizer class to further qualify the resizing options. Of if all you want to stipulate is the handles users have available for resizing you can set the resizeHandles config option with the positions you'd like the resize handles to appear.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    resizeHandles: 'nw ne sw se', // allows resizing of the corners only
    buttons: [{
        text: 'Login'
    }]
});
```

Floating components have a shadow under them to visually emphasize the fact that these components sit above other components in the application. The shadow will appear on the sides and bottom of the window by default. Or, you can specify the shadow config as "sides" in order to show the shadow on all 4 sides or as "drop" to direct the shadow to appear on the bottom-right. The shadowOffset config determines the "height" of the window by setting how far the shadow is from the window element. If a shadow is not the right visual choice for your application you can set shadow: false to prevent the shadow altogether.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    shadow: false, // no shadow is shown for this floating component
    buttons: [{
        text: 'Login'
    }]
});
```

Windows are designed as closable by default. This means that in the window's header there will be a close icon that when clicked will either destroy or hide the window. By default, the window will be destroyed when closed. If your use case for the window requires it to be hidden rather than destroyed you can configure closeAction: 'hide'. By hiding the window you can just show it again later without having to re-create it.

Once a window is shown and focused it may also be closed by pressing the "escape" key. To prevent the escape key from closing the window you can configure the onEsc option with an alternate function or "Ext.emptyFn" to prevent any action from taking place when the escape key is pressed. To prevent a window from being closed by the user at all you can configure it with closable: false.

Note: With the closable config option set to false the user will not see the close icon in the window header and pressing the escape key will have no effect. The window is, however, still closable programmatically using window's close method.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    // closeAction: 'hide', // hides instead of destroys the window when closed
    closable: false, // no close icon is created for this window
    buttons: [{
        text: 'Login'
    }]
});
```

The window API makes it easy to maximize, restore, and minimize a window panel similar to the application windows you may be used to on a desktop machine. The maximize config option displays a maximize button in the panel header that when clicked will cause the window to take up the space of the container / element it's rendered within. Once maximized, the maximize button changes to a restore button that when clicked will restore the window back to the size and position it was prior to being maximized.

Note: The window may also be configured as maximized: true to start the window out as maximized when created.

```
Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    // displays the maximizable tool in the header
    // once maximized the restore tool will swap in with the maximizable tool
    maximizable: true,
    buttons: [{
        text: 'Login'
    }]
});
```

The minimizable config option displays the minimize icon in the header, but itself does not add additional functionality. For the minimize tool to do something when clicked you'll need to add your own logic to the minimize method. In the following example we have a button that when clicked shows the maximizable and minimizable window. Clicking the minimize tool will hide the window.

```
var btn = Ext.create({
    xtype: 'button',
    renderTo: Ext.getBody(),
    text: 'Show User Login',
    handler: function() {
        win.show();
    }
});
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    closable: false,
    animateTarget: btn,
    maximizable: true, // displays the maximize and restore tools in the header
    minimizable: true, // displays the minimize tool in the header
    // performs custom logic when the minimize tool is clicked
});
```

```
minimize: function () {
    this.hide();
},
buttons: [{
    text: 'Login'
}],
});
```

Floating components like window may be re-positioned and re-sized by the user, but they may also be positioned and sized programmatically. You can position a window using the setPosition method.

```
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    buttons: [{
        text: 'Re-position',
        handler: function () {
            // the third param animates the window into the new coordinates
            win.setPosition(0, 0, {
                duration: 250
            });
        }
    }]
});
```

Resizing a window may be accomplished using the setWidth, setHeight, or setSize methods.

```
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    buttons: [{
        text: 'Re-size',
        handler: function () {
            // resizes the window to 2x it's original size
            win.setSize(400, 400);
        }
    }]
});
```

Using the animate method you can get fancy and animate the window into a new position at a new size.

```
var win = Ext.create({
    xtype: 'window',
    title: 'User Login',
    height: 200,
    width: 200,
    autoShow: true,
    buttons: [{
        text: 'Double the window\'s size',
        handler: function () {
            // animate the window to be 2x it's original size
            // and keep it centered at the same time
            win.animate({
                to: {
                    left: win.getX() - (win.getWidth() / 2),
                    top: win.getY() - (win.getHeight() / 2),
                    width: win.getWidth() * 2,
                    height: win.getHeight() * 2
                }
            });
        }
    }]
});
```

```
});
```

### 3.55.2 Conclusion

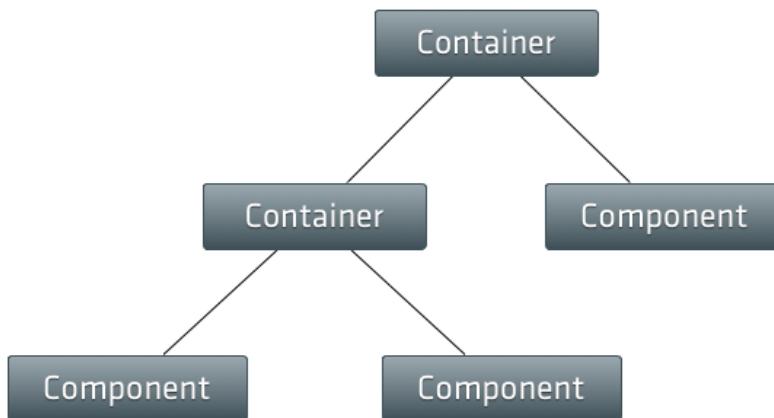
Windows are a handy way to present your end user with additional information without needing to wrestle with the real estate used by the rest of the application. Since windows are a container you can give them any layout fitting your use case. That combined with their floating nature and the ability to maximize and minimize makes windows an extremely versatile component in the Ext JS framework. For light informational popups and confirmation windows check out the Ext.window.MessageBox class.

## 3.56 Layouts and Containers

The layout system is one of the most powerful parts of Ext JS. It handles the sizing and positioning of every Component in your application. This guide covers the basics of how to get started with layouts.

### 3.56.1 Containers

An Ext JS application UI is made up of Components (See the Components Guide for more on Components. A Container is a special type of Component that can contain other Components. A typical Ext JS application is made up of several layers of nested Components.



The most commonly used Container is Panel. Let's take a look at how being a Container allows a Panel to contain other Components:

```
Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    width: 400,
    height: 300,
    title: 'Container Panel',
    items: [
        {
            xtype: 'panel',
            title: 'Child Panel 1',
            height: 100,
            width: '75%'
        },
        {
            xtype: 'panel',
            title: 'Child Panel 2',
            height: 100,
            width: '75%'
        }
    ]
});
```

We just created a Panel that renders itself to the document body, and we used the items config to add two child Panels to our Container Panel.

## 3.57 Layouts

Every container has a layout that manages the sizing and positioning of its child Components. In this section we're going to discuss how to configure a Container to use a specific type of Layout, and how the layout system keeps everything in sync.

### 3.57.1 Using Layouts

In the above example we did not specify a layout for the Container Panel. Notice how the child Panels are laid out one after the other, just as normal block elements would be in the DOM. This happens because the default layout for all Containers is Auto Layout. Auto Layout does not specify any special positioning or sizing rules for child elements. Let's assume, for example, we want our two child Panels to be positioned side by side, and to each take up exactly 50% of the width of the Container - we can use a Column Layout simply by providing a layout config on the Container:

```
Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    width: 400,
    height: 200,
    title: 'Container Panel',
    layout: 'column',
    items: [
        {
            xtype: 'panel',
            title: 'Child Panel 1',
            height: 100,
            columnWidth: 0.5
        },
        {
            xtype: 'panel',
            title: 'Child Panel 2',
            height: 100,
            columnWidth: 0.5
        }
    ]
});
```

To know more about Layouts visit the [Sencha](#) website.

## 3.58 Grids

Ext.grid.Panel is one of the centerpieces of Ext JS. It's an incredibly versatile component that provides an easy way to display, sort, group, and edit data.

### 3.58.1 Basic Grid Panel

Application Users		
Name	Email Address	Phone Number
Lisa	lisa@simpsons.com	555-111-1224
Bart	bart@simpsons.com	555-222-1234
Homer	home@simpsons.com	555-222-1244
Marge	marge@simpsons.com	555-222-1254

Let's get started by creating a basic Ext.grid.Panel. Here's all you need to know to get a simple grid up and running:

#### Model and Store

Ext.grid.Panel is simply a component that displays data contained in a Ext.data.Store. Ext.data.Store can be thought of as a collection of records, or Ext.data.Model instances.

The benefit of this setup is separating our concerns. Ext.grid.Panel is only concerned with displaying the data, while Ext.data.Store takes care of fetching and saving the data using Ext.data.proxy.Proxy.

First, we need to define a Ext.data.Model. A model is just a collection of fields that represents a type of data. Let's define a model that represents a "User":

```
Ext.define('User', {
    extend: 'Ext.data.Model',
    fields: [ 'name', 'email', 'phone' ]
});
```

Next let's create a Ext.data.Store that contains several "User" instances.

```
var userStore = Ext.create('Ext.data.Store', {
    model: 'User',
    data: [
        { name: 'Lisa', email: 'lisa@simpsons.com', phone: '555-111-1224' },
        { name: 'Bart', email: 'bart@simpsons.com', phone: '555-222-1234' },
        { name: 'Homer', email: 'home@simpsons.com', phone: '555-222-1244' },
        { name: 'Marge', email: 'marge@simpsons.com', phone: '555-222-1254' }
    ]
});
```

For sake of ease, we configured Ext.data.Store to load its data inline. In a real world application, you would most likely configure the Ext.data.Store to use an Ext.data.proxy.Proxy to load data from the server.

## Grid Panel

Now, we have a model, which defines our data structure. We have also loaded several model instances into an Ext.data.Store. Now we're ready to display the data using Ext.grid.Panel.

In this example, we configured the Grid with renderTo to immediately render the Grid into the HTML document.

In many situations, the grid will be a descendant of Ext.container.Viewport, which means rendering is already handled.

```
Ext.create('Ext.grid.Panel', {
    renderTo: document.body,
    store: userStore,
    width: 400,
    height: 200,
    title: 'Application Users',
    columns: [
        {
            text: 'Name',
            width: 100,
            sortable: false,
            hideable: false,
            dataIndex: 'name'
        },
        {
            text: 'Email Address',
            width: 150,
            dataIndex: 'email',
            hidden: true
        },
        {
            text: 'Phone Number',
            flex: 1,
            dataIndex: 'phone'
        }
    ]
});
```

And that's all there is to it.

We just created an Ext.grid.Panel that renders itself to the body element. We also told the Grid panel to get its data from the userStore that we previously created.

Finally, we defined the Grid panel's columns and gave them a dataIndex property. This dataIndex associates a field from our model to a column.

The "Name" column has a fixed width of "100px" and has sorting and hiding disabled. The "Email Address" column is hidden by default (it can be shown again by using the menu on any other column header). Finally, the "Phone Number" column flexes to fit the remainder of the Grid panel's total width.

To know more about Grids visit [sencha](#) website.

## 3.59 Forms

A Form Panel is nothing more than a basic Panel with form handling abilities added. Form Panels can be used throughout an Ext application wherever there is a need to collect data from the user.

In addition, Form Panels can use any Container Layout, providing a convenient and flexible way to handle the positioning of their fields. Form Panels can also be bound to a Model, making it easy to load data from and submit data back to the server.

Under the hood a Form Panel wraps a Basic Form which handles all of its input field management, validation, submission, and form loading services. This means that many of the config options of a Basic Form can be used directly on a Form Panel.

### 3.59.1 Basic Form Panel

To start off, here's how to create a simple Form that collects user data:

```
Ext.create('Ext.form.Panel', {  
    renderTo: document.body,  
    title: 'User Form',  
    height: 150,  
    width: 300,  
    bodyPadding: 10,  
    defaultType: 'textfield',  
    items: [  
        {  
            fieldLabel: 'First Name',  
            name: 'firstName'  
        },  
        {  
            fieldLabel: 'Last Name',  
            name: 'lastName'  
        },  
        {  
            xtype: 'datefield',  
            fieldLabel: 'Date of Birth',  
            name: 'birthDate'  
        }  
    ]  
});
```

This Form renders itself to the document body and has three Fields - "First Name", "Last Name", and "Date of Birth". Fields are added to the Form Panel using the items configuration.

The fieldLabel configuration defines what text will appear in the label next to the field, and the name configuration becomes the name attribute of the underlying HTML field.

Notice how this Form Panel has a defaultType of 'textfield'. This means that any of its items that do not have an xtype specified (the "First Name" and "Last Name" fields in this example), are Text Fields.

The "Date of Birth" field, on the other hand, has its xtype explicitly configured as 'datefield', which makes it a Date Field. Date Fields expect to only contain valid date data and come with a DatePicker for selecting a date.

To know more about Forms visit [sencha](#) website.

## 3.60 CORS support

When you are developing the Javascript client separately from the server application, you usually have to deal with cross domain issues. Ajax requests for data are made to a target domain distinct to the original domain where the application code was loaded, and the browser does not allow this.

In the Devon4Sencha there is a solution to resolve this issues:

- Enable request from distinct origin and target domain.
- Security cookies. In the request we need to send the flag `withCredentials` to `true` to say to the browser "We also want send the security cookie in our request". In IE there are some issues about this property that Sencha helps us to manage.

### 3.60.1 Configure CORS support

In the `Config.js` file, you have to define a `CORSEnabled` propertie to a `true` value and add the full path to the server in the `server` property. These are the only things you need to configure to have CORS support. For example:

```
var Config={  
  //...  
  CORSEnabled: true  
  //...  
};
```

### 3.60.2 Bypass CORS security check during development

Sometimes our back-end server is not CORS enabled so we have to deal with it in an alternative way. There are several ways to do it:

- Using a reverse proxy in front of both the client side server and the server side
  - The proxy will fetch the correct files from each server
  - This can be implemented easily with some node.js modules such as indexzero http-server (<https://github.com/indexzero/http-server>)
- Using a browser extension to bypass the CORS policy
  - It is usually easier than the proxy approach although less flexible in what can be accomplished
  - There are several tools available for different browsers

#### Using a browser extension to bypass the CORS policy

Here we present an example for Google Chrome browser called **ForceCORS**.

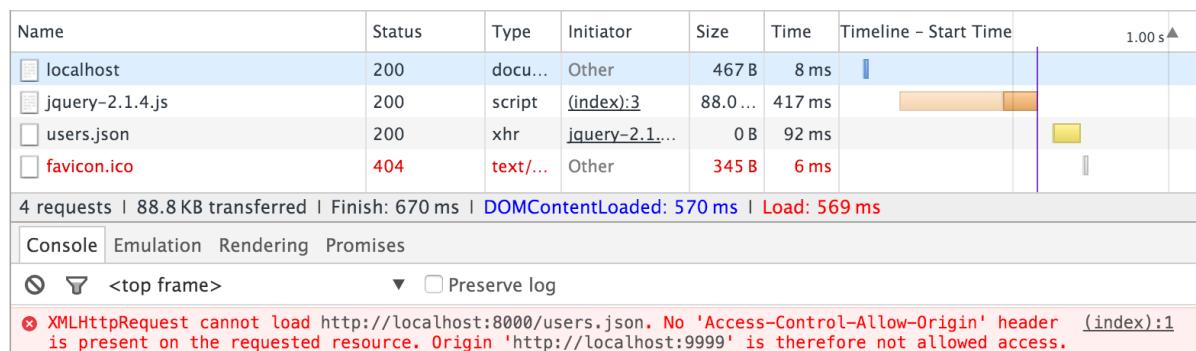
In our example we will use two files: \* `index.html` being served from `localhost:9999` \* `users.json` being served from `localhost:8000`

The `index.html` has a Javascript that makes an AJAX request to fetch the `users.json` file on the other server.

The problem arises because we are trying two things on the requests:

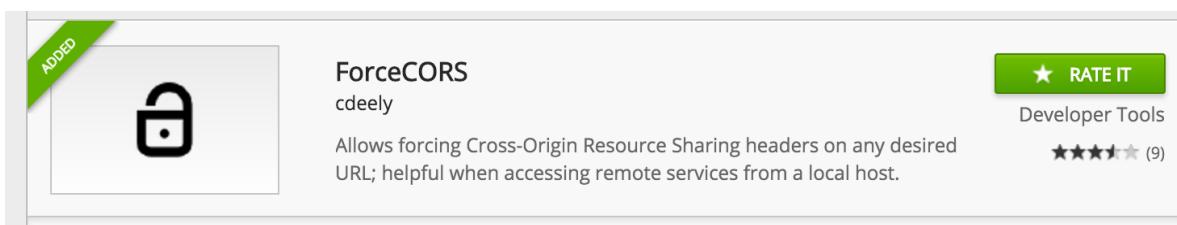
- Accessing a different server through AJAX than the server which served the client code
  - HTML/Javascript files on localhost:9999
  - Backend application in localhost:80000
- Making use of the "with-credentials" setting in the AJAX requests
  - This is important. It means that we want to access localhost:8000 from localhost:9999 AND ALSO send cookies for the localhost:8000 server.
  - If we don't use "with-credentials" then after the login... the next request to localhost:8000 will be sent without cookies... and it will fail on server-side because it has no access!
  - The with-credentials flag is being set on the devon4sencha Ajax.js when Config.CORSEnabled==true

This will result on the page loading and the browser showing the following error on the DevTools console:



To solve it:

1. Install it from the Google WebStore



1. Configure it

## ForceCORS Settings

Below you can specify the desired headers to add or replace per each URL.  
 Use '\*' as a wildcard (which you most likely want at the end of each URL: http://www.something.com/\*)  
 For the full URL filter syntax, see: [https://code.google.com/chrome/extensions/match\\_patterns.html](https://code.google.com/chrome/extensions/match_patterns.html)  
 For a reference of CORS headers, see: <http://www.w3.org/TR/cors/#syntax>  
 You can find the full source code for this extension on GitHub: <https://github.com/chrisdeely/ForceCORS>  
 Icon made by [Freepik](#) from [www.flaticon.com](#)

Add URL

Add

Settings for

http://localhost:8000/\*
 ▼

Header Name	Header Value
Access-Control-Allow-Origin	http://localhost:9999
Access-Control-Allow-Credentials	true

Add Header

Display Intercept Count
Save All
Delete Selected
Delete All
Import/Export

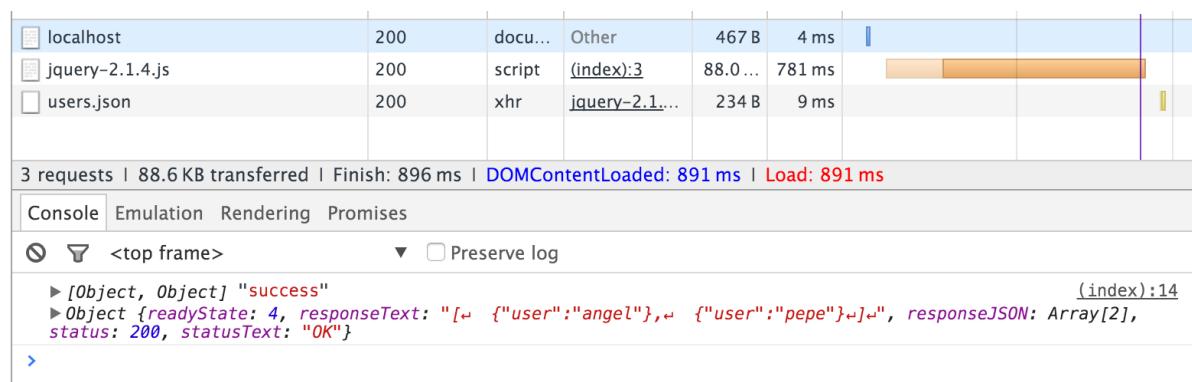
This way we "simulate" that the server side is adding the needed CORS headers to the Ajax response.

### Add more allowed headers

If your client code adds custom headers to the Ajax request (CorrelationId for example in case of an devon4sencha application) then you should add this headers to the **Access-Control-Allow-Headers** configuration setting. For example:

```
Access-Control-Allow-Headers = Origin, X-Requested-With, Content-Type, Accept
```

If we reload the page we should get the data as expected:



## 3.61 Security

Security is an important topic on web applications. JavaScript driven applications must be aware of some patterns to guarantee that information is not compromised.

Following the advice of the [OWASP](#) project, Devon applications take care of:

- [CSRF](#) protection
- Always return JSON with an Object on the outside

The CSRF protection is achieved by introducing a custom header with a value obtained from the server for each request. The token is automatically obtained by `Devon.Security.getCSRF` method on the login process and assigned to each request after that.

The second protection is achieved by using the conventions of the communication protocol between client and server that specifies that every response is an object and effective results are inside a property of this object named `results`.

### 3.61.1 Login process

By default `Devon.App` drives the login process in a configurable way. The steps are:

- Verify existing session by calling `checkExistingSession` on a class with alias `app-security`
- If a session exists: instance a view with an alias of `main-viewport`
- If there is no session: instance a view with an alias of `main-login`

By providing a custom class or views with these aliases the application can introduce its own custom logic or view and take advantage of this login process mechanism.

#### Check existing session

The default implementation for checking the session is provided by `Devon.Security.checkExistingSession` method which makes an ajax request to the `currentuser` REST endpoint on the server.

If the result is successful, the response provides the information for logged user, which is stored on client for later use.

After that, `Devon.App` instantiates the view aliased as `main-viewport` with the current user as data for its `viewModel`.

#### Login window

When there is no existing session in place, the view aliased as `main-login` is instantiated and shown. The default implementation is provided by `Devon.view.login` classes and tries to check user/password with the login endpoint on server.

Upon successful authentication request, the login view must fire an event called `eventLoginSuccessful` that will be trapped by the creator of the view (the `Devon.App` class) to destroy current login form and call the same logic as if an existing session was in place (that is, to instantiate the `main-viewport`)

### 3.61.2 User security info

In the object `Devon.Security.currentUser` we can find information of the current user like `firstname`, `lastname`,... and `roles`. The `roles` object has a value of `true` or `false` for each role name loaded from the server, this way it can be used on several parts of the framework on an easy way as we will see.

```
{  
  currentUser : {  
    login : "name",  
    roles : {  
      ROLE_ADMIN: true,  
      ROLE_MANAGER: true  
    }  
  }  
}
```

### 3.61.3 Controlling visibility of controls based on user roles

Having the current logged user info on the root of the `viewModel` hierarchy (that is, the `viewport`) enables child views to address this info to show/hide controls based on some info. For example:

```
Ext.define('Sample.some.View', {  
  extends : 'Ext.Panel',  
  items : [  
    {  
      html: 'This will be visible always'  
    },  
    {  
      html: 'This is only visible to users with ROLE_ADMIN role',  
      bind:{  
        hidden : '{currentUser.roles.ROLE_ADMIN}'  
      }  
    }]  
});
```

## 3.62 Theming

We are going to create a custom theme for our application. In the application path (devon4sencha/demo) we have to execute the following command in **Sencha Cmd** console:

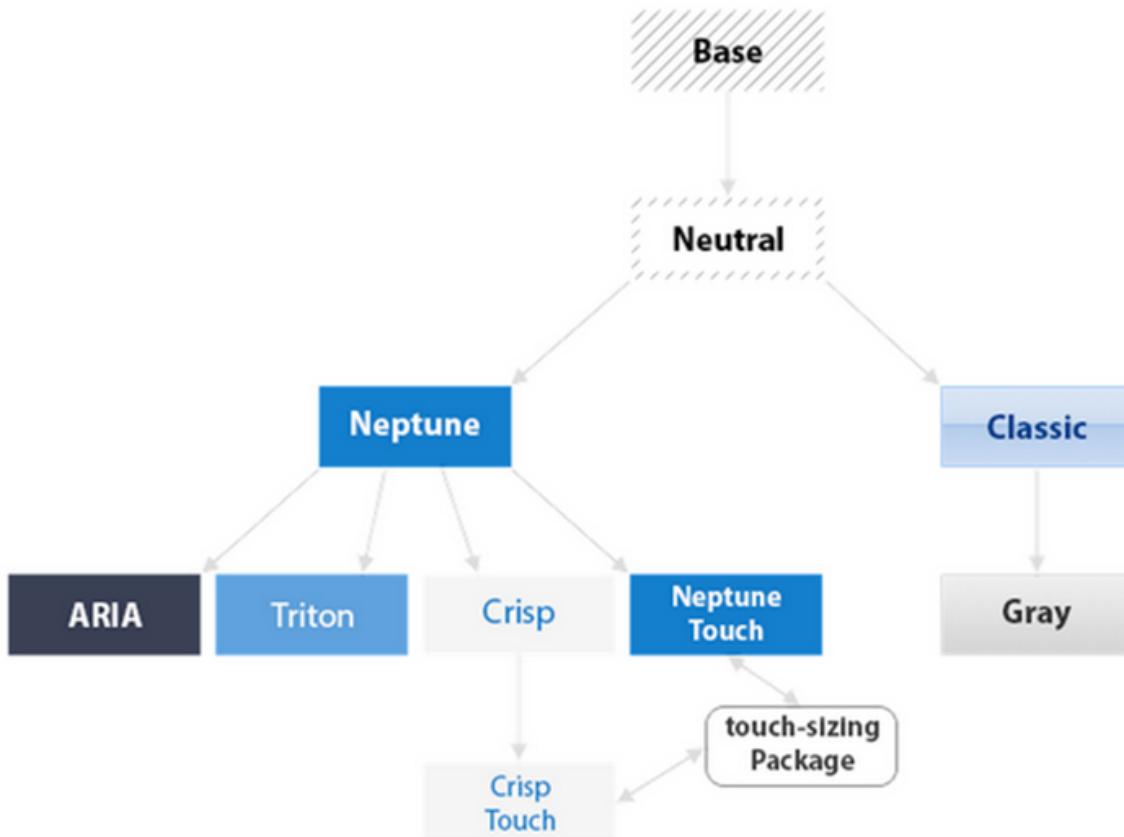
```
sencha generate theme demo-theme
```

This command should have created a new `demo-theme` folder, with the theme files and resources, in the path `devon4sencha/packages/local`. Let's take a look at the default contents of the custom theme folder:

- **package.json** - This is the package properties file. It tells Sencha Cmd certain things about the package like its name, version, and dependencies (other packages that it requires).
- **sass/** - This directory contains all of your theme's Fashion source files. The source files are divided into 4 main sections:
  - **sass/var/** - contains variables.
  - **sass/src/** - contains rules and UI mixin calls that can use the variables defined in `sass/var/`.
  - **sass/etc/** - contains additional utility functions or mixins.
  - **sass/example** - contains files used when generating sample components used in image slicing (despite its name, this folder is used by Sencha Cmd and should not be deleted)
- **resources/** - contains images and other static resources that your theme requires
- **overrides/** - contains any JavaScript overrides to Ext JS component classes that are required for theming those components

### 3.62.1 Configuring Theme Inheritance

All Sencha theme packages are part of a larger hierarchy of themes and each theme package must extend a parent theme. The next step in creating our custom theme is to select which theme to extend. The following image is an overview of Sencha theme hierarchy.



In this tutorial we will create a custom theme that extends the Triton theme. The first step is to configure our custom theme with the name of the theme it is extending. This is done by changing the `extend` property in `packages/local/demo-theme/package.json` from its default value as shown here:

```
"extend": "theme-triton"
```

### 3.62.2 Using a Theme in an Application

To configure your test application to use your custom theme, change the theme selection line in the application `app.json` file in the path `demo/app.json`.

```
"theme": "demo-theme"
```

Run the application and check it the new style. Now the application theme should be like triton theme because we have extended this theme in our **demo-theme**.

### 3.62.3 Making changes in the Theme

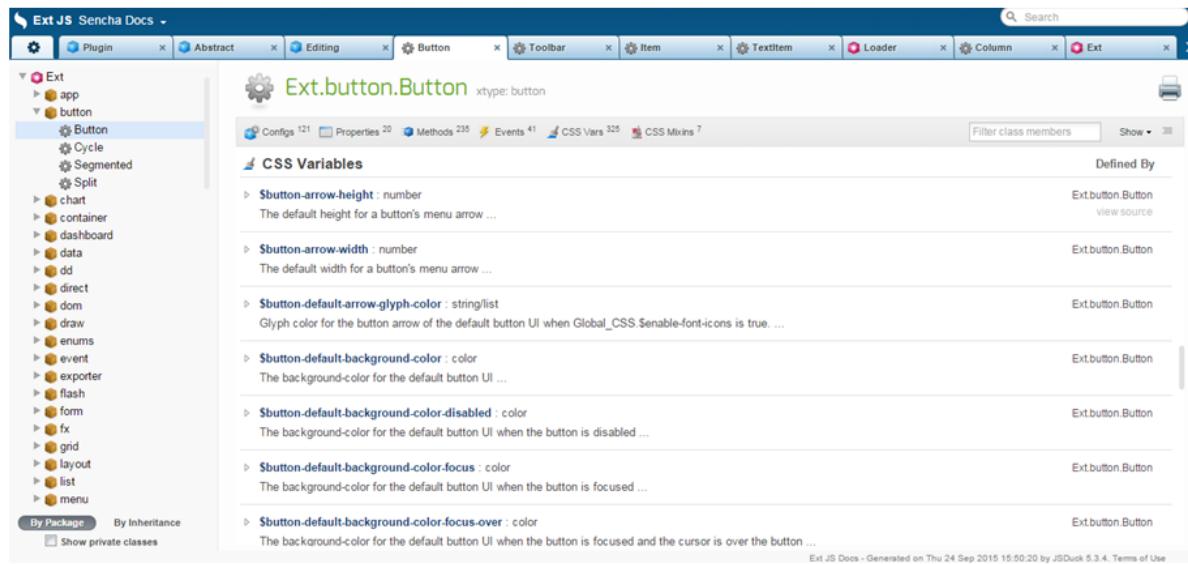
There are a set of variables used across all components. These variables are defined in a global scss class called `Component.scss`. Here is the full documentation for global variables: [https://docs.sencha.com/extjs/6.0/6.0.1-classic#!/api/Global\\_CSS-css\\_var-S-base-color](https://docs.sencha.com/extjs/6.0/6.0.1-classic#!/api/Global_CSS-css_var-S-base-color)

Let's start by modifying the base color from which many Ext JS components' colors are derived. Due to the use of `$base-color` through the default themes making a global change to `$base-color` will have an effect on most all components in the Ext JS library. Create a new folder / file: `packages/local/demo-theme/sass/var/Component.scss`. Add the following code to the `Component.scss` file:

```
$base-color: #317040;
```

Check the new look of our application.

Sencha components style has been defined with CSS variables, so in every component Sencha documentation, there is a section for CSS vars.



The screenshot shows the Ext JS Sencha Docs interface. The left sidebar contains a tree view of Ext JS components, with 'Button' selected. The main panel displays the 'Ext.button.Button' class details. The top navigation bar includes tabs for 'Plugin', 'Abstract', 'Editing', 'Button', 'Toolbar', 'Item', 'Textitem', 'Loader', 'Column', and 'Ext'. The main content area shows the class structure with tabs for 'Configs 121', 'Properties 20', 'Methods 235', 'Events 41', 'CSS Vars 325', and 'CSS Mixins 7'. Below this, the 'CSS Variables' section lists several variables like '\$button-arrow-height', '\$button-arrow-width', and '\$button-default-background-color'. The 'Defined By' column shows that these variables are defined in the 'Ext.button.Button' class. The bottom of the page includes a footer with 'Ext JS Docs - Generated on Thu 24 Sep 2015 15:50:20 by JSDuck 5.3.4. Terms of Use'.

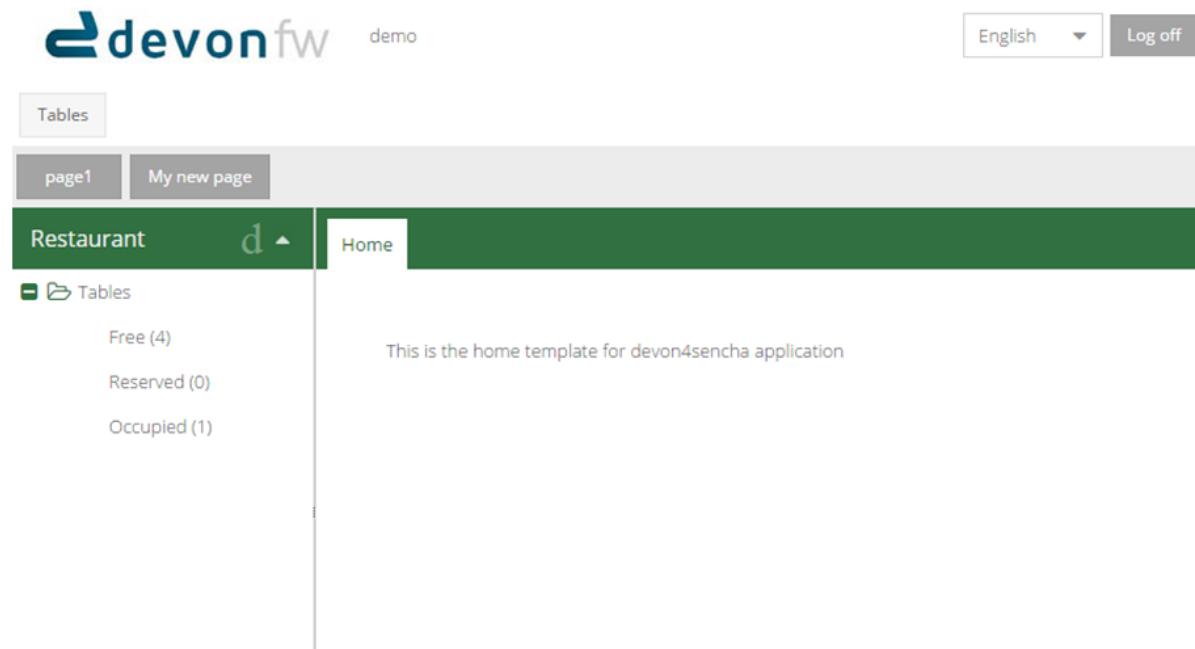
Changing the value of these variables, we can adapt the style of these components to our desired style. For example, we are going to change a bit the style of the buttons.

Create a new folder `button` and a new file `Button.scss`: `packages/local/demo-theme/sass/var/button/Button.scss`.

Add the following code to the `Button.scss` file:

```
$button-default-color: #ffffff;
$button-default-base-color: #A4A4A4;
```

Go to the application and check the grey background-color and the white color of the `page1` and `My new page` toolbar buttons.



The screenshot shows the devonfw application interface. The top navigation bar includes a logo, the text 'demo', a language selector ('English'), and a 'Log off' button. The main content area features a sidebar with a 'Tables' tab. Below the sidebar, there are two tabs: 'page1' (selected) and 'My new page'. The main content area has a green header bar with 'Restaurant' and 'Home' buttons. The left sidebar shows a list of tables: 'Free (4)', 'Reserved (0)', and 'Occupied (1)'. The main content area displays the message: 'This is the home template for devon4sencha application'.

### 3.62.4 Adding customs UI's

Every component in the Ext JS framework has a user interface (ui) configuration, which defaults to default. This property can be configured on individual component instances to give them a different appearance from other instances of the same type. For example, in the Neptune theme, panels with the 'default' UI have dark blue headers and panels with the 'light' UI have light blue headers.

The theme-neutral theme includes mixins for many different Ext JS components. You can call these mixins to generate new UIs for components. Available mixins for each component are listed in the API documentation. For example, see Ext.button.Button in Sencha documentation and scroll down to the CSS Mixins section to see what parameters the Button UI mixin accepts.

We are going to create a custom UI for a small button in toolbar.

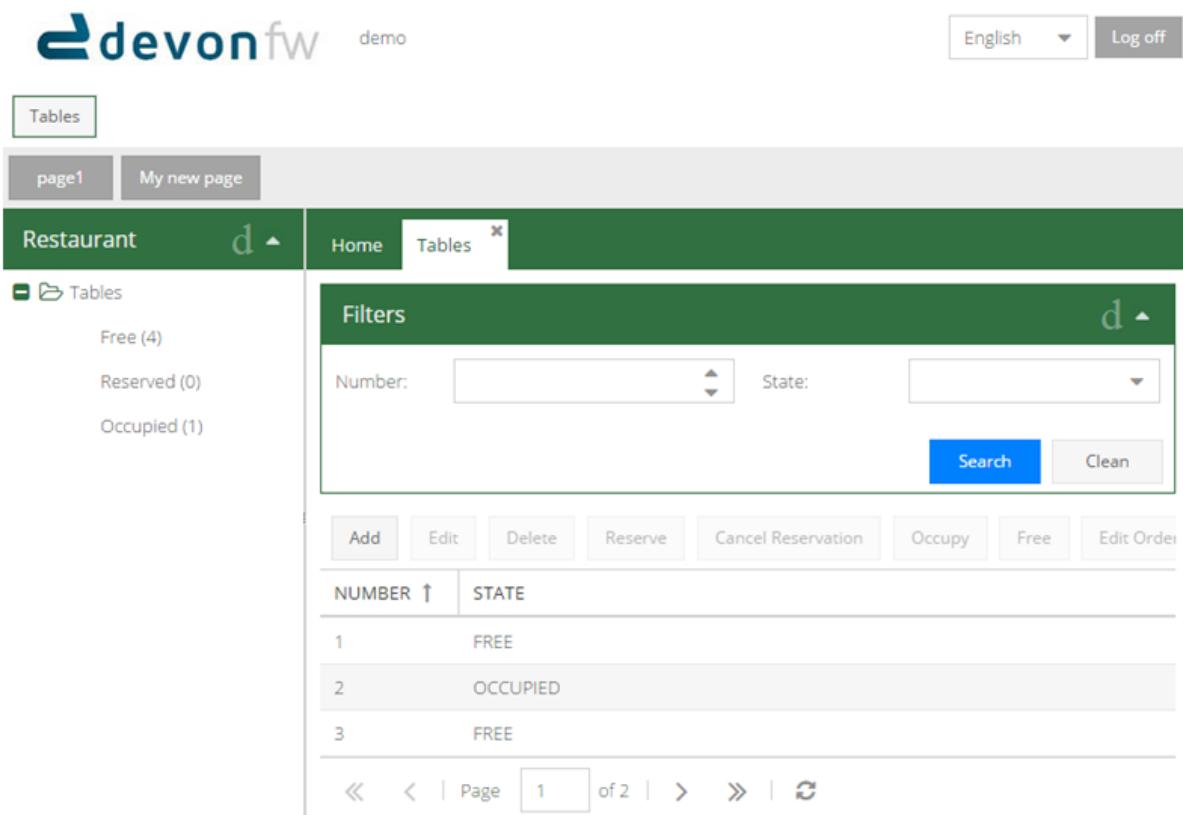
Create a new folder `button` inside `scr` and a new file: `packages/local/demo-theme/sass/src/button/Button.scss`. Add the following code to the `Button.scss` file:

```
@include extjs-button-toolbar-small-ui(  
  $ui: 'blue-btn',  
  
  $border-color: transparent,  
  $border-color-over: transparent,  
  $border-color-focus: transparent,  
  $border-color-pressed: transparent,  
  $border-color-focus-over: transparent,  
  $border-color-focus-pressed: transparent,  
  $border-color-disabled: transparent,  
  
  $background-color: #0080FF,  
  $background-color-over: #0080FF,  
  $background-color-focus: #0080FF,  
  $background-color-pressed: #0080FF,  
  $background-color-focus-over: #0080FF,  
  $background-color-focus-pressed: #0080FF,  
  $background-color-disabled: #0080FF,  
  
  $color: #ffffff,  
  $color-over: #ffffff,  
  $color-focus: #ffffff,  
  $color-pressed: #ffffff,  
  $color-focus-over: #ffffff,  
  $color-focus-pressed: #ffffff,  
  $color-disabled: #ffffff  
)
```

By default, Sencha apply the CSS class `x-btn-default-toolbar-small` to a normal button in a toolbar so as we said the default ui for this kind of buttons is `default`, for this reason, we have to change the button ui to use our `blue-btn` style. Add button ui property in the Search button in the file `TableListV.js`.

```
{  
  xtype: 'button',  
  ui: 'blue-btn-toolbar',  
  text: 'Search',  
  width: 80,  
  listeners: {  
    click: 'doSearch'  
  }  
}
```

Check the result.



The screenshot shows a web-based application interface for managing table reservations. At the top, there is a navigation bar with the 'devonfw' logo, a 'demo' link, a language selector ('English'), and a 'Log off' button. Below the navigation bar, there is a breadcrumb trail: 'Restaurant' > 'Tables' > 'Tables'. The main content area is titled 'Filters' and contains two dropdown menus: 'Number:' and 'State:', with 'Search' and 'Clean' buttons below them. A table below the filters lists three table entries: '1 FREE', '2 OCCUPIED', and '3 FREE'. At the bottom of the table, there are navigation buttons for 'Page' (with '1' selected), 'of 2', and various actions: 'Add', 'Edit', 'Delete', 'Reserve', 'Cancel Reservation', 'Occupy', 'Free', and 'Edit Order'.

NUMBER ↑	STATE
1	FREE
2	OCCUPIED
3	FREE

## 3.63 Error processing

Devon's Sencha layer has its own mechanism to manage received errors from the server when performing Ajax requests. Devon has two features for related to error management:

- Show an alert message for errors not processed by the application
- Show an error message in forms fields

First here it is an example of an error message received from server when validation exception occurs:

```
{  
  "message" : "{number=[must be less than or equal to 1000]}",  
  "errors":{  
    "tablenumber": [  
      "must be less than or equal to 1000"  
    ]  
  },  
  "uuid": "55293fcd-ff38-4440-ad4c-8cd48eec7cac",  
  "code": "ValidationException"  
}
```

The errors key has the messages for every field that has errors in a view. In this example case, the field with errors is tablenumber.

Having standardized this format for Devon applications allows us to create automatic processing of messages. By using Devon REST endpoints to accessing the server, if we receive in the JSON response the field errors, Devon will manage the exception as a form validation exception, and if we don't received the field errors, Devon will show a default alert message.

In case the application wants to manage the processing of the errors in its own way, it should provide a failure property to the REST endpoint method call.

In the case that we received the field errors, Devon will perform a search in the current view for a component with the same reference property as the error key (reference it is a Sencha property to distinguish a component from other component) and it will mark as invalid the component, showing the message received alongside.

In our example, Devon will find a component that has as a reference property with value "number" and it will mark it as invalid. By default, Devon starts searching for components to match with the error validation key from the top of the view, although you can tell Devon where to start the search with the property referenceView. The value of the property referenceView also has to be the value of a view component reference property.

### SampleViewController.js.

```
Devon.rest.tablemanagement.table.post({  
  scope: this,  
  jsonData : vm.get('table'),  
  referenceView: 'panel',  
  success: this.tableCrudClose  
});
```

Of course, you can override Devon exception management defining a failure function. In that case, when the server responds with an error, Devon will call user failure function and user should manage the error.

### SampleViewController.js.

```
Devon.rest.tablemanagement.table.post({
  scope: this,
  jsonData : vm.get('table'),
  success: this.tableCrudClose,
  failure: this.myFailureFn,
});
```

## 3.64 Internationalization

Internationalization in Devon applications is performed independently on the client and the server side. On the client side, literal JavaScript objects are used to define the messages similar to the properties files used on the server side.

As messages are used during the definition of Sencha classes for the application, these have to be loaded prior to their usage. Devon4Sencha takes care of this by following the next steps.

### 3.64.1 Configuration

Firstly, you need to configure the default language used for the application when loading and the group of possible languages the application can make use of.

Config object is used to set the following properties:

#### Config.js.

```
supportedLocales = ['en_EN', 'de_DE', 'es_ES'];
defaultLocale = 'en_EN';
```

- **supportedLocales** (Array[String]): List of application supported locales in ISO format (language\_country\_variant).
- **defaultLocale** (String): Default application locale.

### 3.64.2 Message bundles

Message bundles are group of key/value pairs that defines the literals for a language in a hierarchical way by using the JavaScript object literal format. In Devon Framework, a language bundle is a singleton class that inherits from **Devon.I18nBundle** class. The class name, has to end with the corresponding locale code:

```
/*en_EN bundle for Table*/
Ext.define('Sample.view.table.i18n.Table_en_EN', {
    extend:'Devon.I18nBundle',
    singleton:true,
    i18n:{
        tables: {
            title: 'Tables',
            html:'List of tables for the restaurant demo'
        },
        only_en_EN:'Only in english'
    }
});

/*es_ES bundle for Table*/
Ext.define('Sample.view.table.i18n.Table_es_ES', {
    extend:'Devon.I18nBundle',
    singleton:true,
    i18n:{
        tables: {
            title: 'Mesas',
            html:'Lista de mesas del restaurante de demostración'
        }
    }
});
```

Note that, all the language key/values are inside an **i18n** property object, that can be used in another Sencha class by referencing a global **i18n** variable as follows:

```
Ext.define("Sample.view.table.TableList", {
    extend: "Ext.panel.Panel",
    title: i18n.tables.title,
    html: i18n.tables.html
});
```

Therefore, a view needs all the referenced bundles to get loaded before loading itself. As a recommendation, these bundles can be loaded in a main controller by requiring the message bundle object prior to the view object:

```
Ext.define('Sample.controller.table.TablesController', {
    extend: 'Ext.app.Controller',
    requires:[
        'Sample.view.table.i18n.Table_en_EN',
        'Sample.view.table.i18n.Table_es_ES',
        'Sample.view.table.TableList'
    ]
});
```

A language tag has to be defined at least for the default locale bundle. As shown in the sample, the tag `i18n.only_en_EN` is only defined in the bundle `en_EN`, so when the language is set to the other possible language `es_ES`, this tag has the value "*Only in english*".

### 3.64.3 Change language

The current locale for the application can be changed with the static method `Devon.I18n.setCurrentLocale((String)locale)`. After the method executes, the global object `i18n` points to the specified locale. Devon framework saves this locale in a cookie, and the page is reloaded with `location.reload` showing the changes:

```
Ext.define('Sample.view.main.HeaderVC', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.main-header',

    languageChange : function(combo){
        Devon.I18n.setCurrentLocale(combo.getValue());
        location.reload();
    }
});
```

## 4. Devon4Sencha Cookbook

### 4.1 Mocks with Simlets: simulating server responses

Sencha Ext JS provides a utility to simulate server responses to any request, right in the browser.

Generally, these utilities are called `simlets`. Mocking something is not unknown to developers who write unit tests, however mocking an entire server or even a part of it isn't widely known and used during the client application development.

When developing client-server applications, the client, especially a single page application or a native client is designed to be independent from the server or it should really be.

Sencha out-of-box provides XML and JSON type of AJAX simlets, base classes are available to extend and create new ones that support other type of requests or envelopes, such as multipart/form-data or remote procedure calls.

#### 4.1.1 Simlet Service

A singleton manager class provides transparent service to catch all the requests being sent to a specific endpoint without having the developer to change the application code. Endpoint doesn't even have to be existing, again, no running server required. See `Ext.ux.ajax.SimManager` for further details.

#### 4.1.2 JSON Simlet

First, the simlet is registered. The following one define an endpoint, that will catch all requests sent to `/users` then returns an array of objects in JSON format.

```
Ext.ux.ajax.SimManager.register({
  url : 'users',
  type : 'json',
  data : [
    {
      id : 1,
      username : 'server.simon',
      name : 'Server Simon'
    },
    {
      id : 2,
      username : 'client.clare',
      name : 'Client Clare'
    }
  ]
});
```

Now, we create an AJAX GET request to the `/user's` endpoint. It is simple as follows:

```
Ext.Ajax.request({
  url : 'users',
  method : 'GET',
  success : function(response) {
    var users = Ext.decode(response.responseText);

    console.log(users.length); // 2
  }
});
```

The above simlet is static and doesn't distinguish request methods, however, it's often required to handle different methods and more complex requests. This is possible by registering dynamic simlets:

```
Ext.ux.ajax.SimManager.register({
    url : 'users',
    type : 'json',
    data : function (request) {
        var xhr = request.xhr,
            requestMethod = xhr.method,
            requestHeaders = xhr.requestHeaders,
            requestBody = Ext.decode(xhr.body);

        // custom logic goes here

        return [
            {
                id : 1,
                username : 'server.simon',
                name : 'Server Simon'
            },
            {
                id : 2,
                username : 'client.clare',
                name : 'Client Clare'
            }
        ];
    }
});
```

Inside the method, every detail of the request is available for further processing, response has to be returned here as well.

## 4.2 Simlets in Devon4Sencha

The Sample Application `ExtSample` comes with the Simlets configuration ready to be used. Inside the `app` folder of the sample application, there is a file called `Simlets.js` that contains the server responses simulated for the AJAX requests:

```
Ext.define('Sample.Simlets', {
    singleton: true,
    requires:[
        'Ext.ux.ajax.SimManager', 'Ext.ux.ajax.JsonSimlet'
    ],

    useSimlets : function(){
        Ext.ux.ajax.SimManager.init({
            defaultSimlet: 404,
            defaultType: 'json'
        });
        Ext.ux.ajax.SimManager.register({
            url:/currentuser/,
            data: { "id":2,"name":"waiter","firstName":"Willy","lastName":"Waiter","role":"WAITER" }
        });
        Ext.ux.ajax.SimManager.register({
            url:/csrf-token/,
            data: { headerName: 'FAKE-CSRF', token: '1234-fake' }
        });
        Ext.ux.ajax.SimManager.register({
            url:/login/,
            data: { headerName: 'FAKE-CSRF', csrf: '1234-fake' }
        });
        Ext.ux.ajax.SimManager.register({
            url:/tablemanagement.v1.table.search/,
            data: function(request){
                console.log(Ext.decode(request.xhr.body));

                return { "pagination":{ "size":500, "page":1, "total":null }, "result": [
                    { "id":101, "modificationCounter":1, "revision":null, "waiterId":null, "number":1, "state":"OCCUPIED" },
                    { "id":102, "modificationCounter":2, "revision":null, "waiterId":null, "number":2, "state":"OCCUPIED" },
                    { "id":103, "modificationCounter":1, "revision":null, "waiterId":null, "number":3, "state":"FREE" }
                ] };
            }
        });
    }
});
```

```

  {"id":104,"modificationCounter":1,"revision":null,"waiterId":null,"number":4,"state":"FREE"},  

  {"id":105,"modificationCounter":1,"revision":null,"waiterId":null,"number":5,"state":"FREE"}]};  

  }  

});  

Ext.ux.ajax.SimManager.register({  

  url:/tablemanagement/v1/table//,  

  data: function(request){  

    console.log(Ext.decode(request.xhr.body));  

    console.log("simlet used");  

    return  

  {"id":101,"modificationCounter":1,"revision":null,"waiterId":null,"number":1,"state":"OCCUPIED"}  

  }  

});  

Ext.ux.ajax.SimManager.register({  

  url:/salesmanagement/v1/orderposition/,  

  data: function(request){  

    console.log(Ext.decode(request.xhr.body));  

    return  

  [{"id":1,"modificationCounter":1,"revision":null,"orderId":1,"cookId":null,"offerId":1,"offerName":"Schnitzel-  

  Menü","state":"DELIVERED","drinkState":"DELIVERED","price":6.99,"comment":"mit Ketschup"},  

  {"id":2,"modificationCounter":1,"revision":null,"orderId":1,"cookId":null,"offerId":2,"offerName":"Goulasch-  

  Menü","state":"DELIVERED","drinkState":"DELIVERED","price":7.99,"comment":""},  

  {"id":3,"modificationCounter":1,"revision":null,"orderId":1,"cookId":null,"offerId":3,"offerName":"Pfifferlinge-  

  Menü","state":"DELIVERED","drinkState":"DELIVERED","price":8.99,"comment":""},  

  {"id":4,"modificationCounter":1,"revision":null,"orderId":1,"cookId":null,"offerId":4,"offerName":"Salat-  

  Menü","state":"DELIVERED","drinkState":"DELIVERED","price":5.99,"comment":""}];  

  }  

});  

});  

});
  
```

In the file `Application.js,` the Simlets file has been added as a required source and in the launch section the simlets have been initialized:

```

Ext.define('Sample.Application', {  

  extend: 'Devon.App',  

  name: 'Sample',  

  requires:[  

    'Sample.Simlets'  

  ],  

  controllers: [  

    'Sample.controller.main.MainController',  

    'Sample.controller.table.TablesController',  

    'Sample.controller.cook.CookController'  

  ],  

  launch: function() {  

    Devon.Log.trace('Sample.app launch');  

    console.log('Sample.app launch');  

    if (document.location.toString().indexOf('useSimlets')>=0){  

      Sample.Simlets.useSimlets();  

    }  

    this.callParent(arguments);  

  }
});
  
```

Finally, in the app.json, the package ux has been added (just above devon-extjs) for loading Ext.ux Simlet classes:

```

  "requires": [  

    "font-awesome",  

    "devon-extjs",
  ]
  
```

1,

Therefore, for launching the ExtSample application, you just have to type the following url in the browser:

```
http://localhost:1841/ExtSample/?useSimlets=true
```

## 4.2.1 Use Simlets in your client application

Basically, you need to follow the same steps as for the sample application:

1. Create the file `Simlets.js` for specifying the AJAX requests you want to simulate.
2. In `Application.js`, add the requires for the simlets file and the configuration for launching them.
3. In `app.json`, include the `ux` package in the requires, for loading `Ext.ux Simlets` classes.
4. Launch your application, by adding `?useSimlets=true` to the url: <http://localhost:1841/MyApp/?useSimlets=true>

## 4.3 Simlets Benefits

Using Simlets, you can develop your client app without depending on the development speed on the server side. Sometimes, you can observe, how projects get delayed because some developers are waiting for some data from the server to test their code.

Here is a not so imaginary conversation between two developers:

**Clare (client):** Hey dude, I really need you to make that API work.

**Simon (server):** Yeah, I'm working on it, give me few more days!

**Clare (client):** No way buddy, I cannot go any further with my task, you're holding me up!

**Simon (server):** I'm doing my best. I'll let you know.

Unfortunately, Simon did not get back to Clare in time and the project got delayed. Have you ever found yourself in the same situation?

Using Simlets. Now Clare could simulate an API to go further with her task and no need to wait until Simon finishes his one.

## 4.4 Devon4Sencha best practices

### 4.4.1 Bad formed code

- **Trailing coma**

- Not every browser support it
- Difficult to debug

```
var myObject = {  
    foo: 1,  
    bar: 'value',  
    traillingComma: true,  
};
```

```
var me = this,  
    store = me.getStore(),  
    firstRec = store.getAt(0) //--- missing comma  
accidentalGlobal = true;
```

- Unused function parameters should be removed.
- Functions should not be too complex.
- Unused local variables should be removed.
- Control structures should use always curly brackets.
- Variables should not be redeclared.
- Each statement should end with a semicolon.
- Nested blocks of code should not be left empty.
- Sections of code should not be "commented out".

### 4.4.2 Excessive or unnecessary nesting of component structures

- One of the most common mistakes developers make is nesting components for no reason.
- Doing this hurts performance and can also cause oddities such as double borders or unexpected layout behavior.
- **Example:** Panel that contains a single grid. Panel is unnecessary. The extra panel can be eliminated. Forms, trees, tab panels and grids all extend from Panel, so you should especially watch for unnecessary nesting conditions

```
items: [ {  
    xtype: 'panel',  
    title: 'My Cool Grid',  
    layout: 'fit',  
    items: [ {  
        xtype: 'grid',  
        store: 'Mystore',  
    } ] } ]
```

```

        columns: [{...
    }]
}

}]

```

Example 1A BAD: The panel is unnecessary.

```

layout: 'fit',
items: [{{
    xtype: 'grid',
    title: 'My cool Grid',
    store: 'MyStore',
    columns: [{...
    }]
}
}]

```

Example 1B Good: The grid is already a panel, so just use any panel properties directly on the grid.

#### 4.4.3 Memory leaks caused by failure to cleanup unused components

- **Why my app is getting slower and slower?**
- **Failure to cleanup unused components as user navigates**
  - Each time user right-clicks on a grid row a new context menu is created
  - If the user keeps this app open and clicks hundreds of times, hundreds of context menus will never be destroyed
  - To the developer and user, the apps "looks" visually correct because only the last context menu created is seen on the page
  - Memory utilization will keep increasing
  - Slower operation / browser crash

```

Ext.define('MyApp.view.MyGrid', {
    extend: 'Ext.grid.Panel',
    columns: [{...}],
    store: 'MyStore',
    initComponent: function() {
        this.callParent(arguments);
        this.on({
            scope: this,
            itemcontextmenu: this.onItemContextMenu
        });
    },
    onItemContextMenu: function(view, rec, item, index, event) {
        event.stopEvent();
        Ext.create('Ext.menu.Menu', {
            items: [{{
                text: 'Do Something'
            }]
        }).showAt(event.getXY());
    }
}

```

```
});
```

- **Better solution**

- Context menu is created once when grid is initialized and is simply reused each time
- However, if the grid is destroyed, the context menu will still exist

```
Ext.define('MyApp.view.MyGrid', {
    extend: 'Ext.grid.Panel',
    columns: [{...}],
    store: 'MyStore',
    columns: [{...}],
    initComponent: function() {
        this.menu = this.buildMenu();
        this.callParent(arguments);
        this.on({
            scope: this,
            itemcontextmenu: this.onItemContextMenu
        });
    },
    buildMenu: function() {
        return Ext.create('Ext.menu.Menu', {
            items: [{{
                text: 'Do something'
            }}]
        });
    },
    onItemContextMenu: function(view, rec, item, index, event) {
        event.stopEvent();
        Ext.create('Ext.menu.Menu', {
            items: [{{
                text: 'Do Something'
            }}]
        }).showAt(event.getXY());
    }
});
```

- **Best solution**

- Context menu is destroyed when the grid is destroyed

```
Ext.define('MyApp.view.MyGrid', {
    extend: 'Ext.grid.Panel',
    columns: [{...}],
    store: 'MyStore',
    columns: [{...}],
    initComponent: function() {
        this.menu = this.buildMenu();
        this.callParent(arguments);
        this.on({
            scope: this,
            itemcontextmenu: this.onItemContextMenu
        });
    },
});
```

```
buildMenu: function() {
    return Ext.create('Ext.menu.Menu', {
        items: [{
            text: 'Do something'
        }]
    });
},

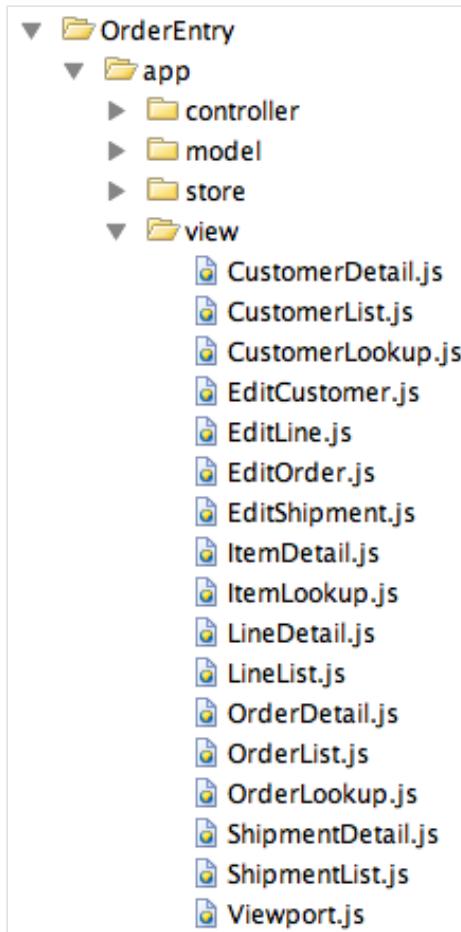
onDestroy: function() {
    this.menu.destroy();
    this.callParent(arguments);
},

onItemContextMenu: function(view, rec, item, index, event) {
    event.stopEvent();
    Ext.create('Ext.menu.Menu', {
        items: [{
            text: 'Do Something'
        }]
    }).showAt(event.getXY());
}
});
```

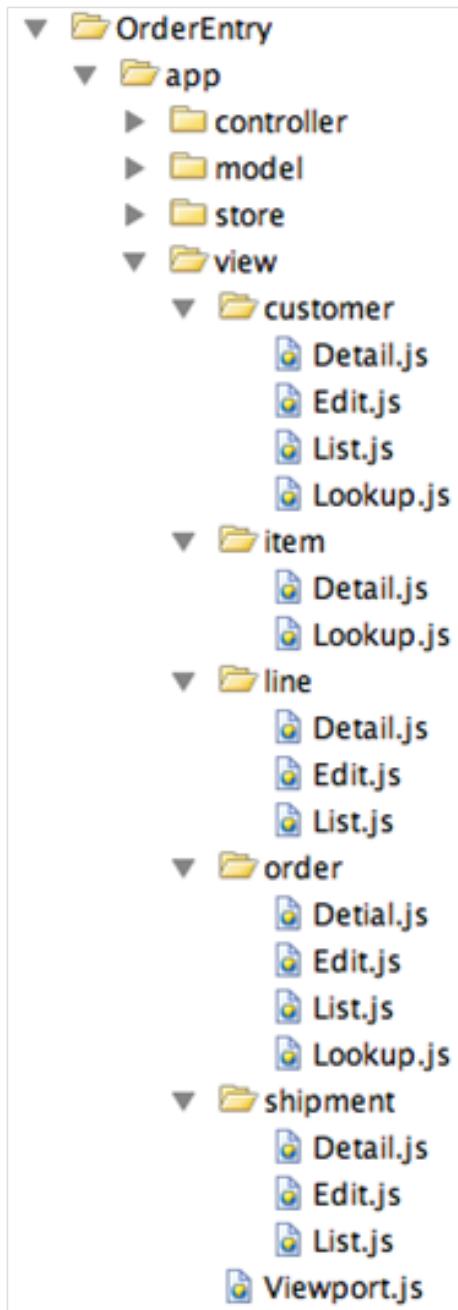
#### 4.4.4 Poor folder structure for source code

- Doesn't affect performance or operation, but it makes it difficult to follow the structure of the app

Example 1A BAD: Poor folder structure:



Example 1B Good: Folder structure to follow:



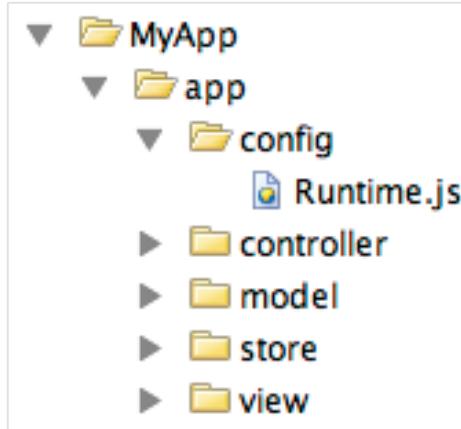
#### 4.4.5 Use of global variables

- Name collisions and hard to debug.
- We should hold "properties" in a class and then reference them with getters and setters.

Instead of

```
MyLastCustomer= 123456;
```

We use



```
Ext.define('MyApp.config.Runtime', {
    singleton: true,
    config: {
        myLastCustomer: 0 // initialize to 0
    },
    constructor: function(config) {
        this.initConfig(config);
    }
});
```

```
MyApp.config.setMyLastCustomer(12345);

MyApp.config.getMyLastCustomer();
```

#### 4.4.6 Use of *id*

- Use of id's on components is not recommended.
- Each id must be unique.

```
//here we define the first save button
xtype: 'toolbar',
    items: [{
        text: 'Save Picture',
        id: 'savebutton'
    }]

//somewhere else in the code we have another component with an id of 'savebutton'

xtype: 'toolbar',
    items: [{
        text: 'Save Order',
        id: 'savebutton'
    }]
```

- Replace by "itemId" resolves the name conflict and we can still get a reference to the component.

```
xtype: 'toolbar',
    itemId: 'picturetoolbar',
    items: [{
        text: 'Save Picture',
        itemId: 'savebutton'
    }]

// somewhere else in the code

xtype: 'toolbar',
```

```
itemId: 'ordertoolbar',
items: [
  text: 'Save Order',
  itemId: 'savebutton'
}]
```

```
var pictureSaveButton = Ext.ComponentQuery.query('#picturetoolbar > #savebutton')[0];
var orderSaveButton = Ext.ComponentQuery.query('#ordertoolbar > #savebutton')[0];
// assuming we have a reference to the "picturetoolbar" as picToolbar
picToolbar.down('#savebutton');
```

#### 4.4.7 Unreliable referencing of components

- Code that relies on component positioning in order to get a reference.
- It should be avoided as the code can easily be broken if any items are added, removed or nested within a different component.

```
var mySaveButton = myToolbar.items.getAt(2);
var myWindow = myToolbar.ownerCt;
```

```
var mySaveButton = myToolbar.down('#savebutton');
var myWindow = myToolbar.up('window');
```

#### 4.4.8 Failing to follow upper or lowercase naming conventions

- Avoid confusion and keep your code clean.
- Additionally, if you are firing any custom events, the name of the event should be all lowercase.

Wrong upper lower naming convention

```
Ext.define('MyApp.view.customerList', {
  extend: 'Ext.grid.Panel',
  alias: 'widget.CustomerList',
  MyCustomConfig: 'xyz',
  initComponent: function() {
    Ext.apply(this, {
      store: 'Customers',
      ...
    });
    this.callParent(arguments);
  }
});
```

Correct upper lower naming convention

```
Ext.define('MyApp.view.CustomerList', {
  extend: 'Ext.grid.Panel',
  alias: 'widget.customerList',
  MyCustomConfig: 'xyz',
  initComponent: function() {
    Ext.apply(this, {
      store: 'Customers',
      ...
    });
    this.callParent(arguments);
  }
});
```

```
});
```

#### 4.4.9 Making your code more complicated than necessary.

- Each value is loaded individually

```
//suppose the following fields exist within a form
items: [
  {
    fieldLabel: 'User',
    itemId: 'username'
  },
  {
    fieldLabel: 'Email',
    itemId: 'email'
  },
  {
    fieldLabel: 'Home Address',
    itemId: 'address'
  }
];

//you could load the values from a record into each form field individually

myForm.down('#username').setValue(record.get('UserName'));
myForm.down('#email').setValue(record.get('Email'));
myForm.down('#address').setValue(record.get('Address'));
```

- Use "loadRecord" method

- Review all of a component's methods and examples to make sure you are using simple and proper techniques.

```
items: [
  {
    fieldLabel: 'User',
    name: 'UserName'
  },
  {
    fieldLabel: 'Email',
    name: 'Email'
  },
  {
    fieldLabel: 'Home Address',
    name: 'Address'
  }
];

myForm.loadRecord(record);
```

#### 4.4.10 Nesting callbacks are a nightmare

- Pyramidal code
- Will cost problems in the future
- Difficult to
  - Read
  - Comprehend
  - Follow
  - Debug

```
Ext.Ajax.request({
  url: 'someUrl.php',
  success: function(response) {
    // Do work here
    Ext.Ajax.request({
      url: 'anotherUrl.php',
```

```
success: function(response) {
    // Do more work here
    Ext.Ajax.request({
        url: 'yetAnotherUrl.php',
        success: function(response) {
            // Do yet more work here
            Ext.Ajax.request({
                url: 'yetAnotherUrl.php',
                success: function(response) {
                    // This is pretty ridiculous.
                }
            });
        }
    });
});
```

- Use "scope"

```

getPeople: function(people) {
    Ext.Ajax.request({
        url: 'people.php',
        method: 'GET',
        params: people,
        scope: this,
        success: this.onAfterGetPeople
    });
}

onAfterGetPeople: function(response) {
    // Do some work here

    var jsonData = Ext.decode(response.responseText);
    this.getDepartments(jsonData.departments);
}

getDepartments: function(departments) {
    Ext.Ajax.request({
        url: 'departments.php',
        method: 'GET',
        params: 'departments',
        scope: this,
        success: this.onAfterGetDepartments
    });
}

onAfterGetDepartments: function(response) {
    // Do more work
}
}

```

#### 4.4.11 Caching and references

- Wrong use of object references.
    - Loop accessing an object.
    - Repetition of accessing to the object.
    - Use references!
      - Store object in a variable.
      - Improvement of application performance.

- Avoid using:
  - `document.getElementById()`
  - `Ext.getCmp()`
  - and other global queries.
- `jsPerf`
  - JavaScript performance playground.
  - Aims to provide an easy way to create and share test cases, comparing the performance of different JavaScript snippets by running benchmarks.
  - <http://jsperf.com>

```

for (var i = 0; i < 1000; i++) {
  globalVar.some.ridiculous.chain.method();
  globalVar.some.ridiculous.chain.value = 'foobar';

// you get the idea
}
  
```

```

var localReference = globalVar.some.ridiculous.chain;

for (var i = 0; i < 1000; i++) {
  localReference.method();
  localReference.value = 'foobar';

//you get the idea
}
  
```

#### 4.4.12 Indentation

- Code impossible to follow
  - Too much!
  - Inconsistent
  - Waste

##### Improperly indented & wasteful code

Too much!      Inconsistent      Waste

```

7  segmentedContactAlphabetical: function(segButton, button, pressed) {
8    Ext.getStore('ContactListStore').clearFilter();
9    if(pressed) {
10      Ext.getStore('ContactListStore').filterBy( function(record) {
11        if( record.getName().indexOf(button.getText()) == 0 ) {
12          return record;
13        }
14      });
15
16      if(Ext.getStore('ContactListStore').getCount() == 0){
17        this.getContactDetailView().setActiveItem(0);
18      }else{
19        var records=this.getContactView().query('#listContactItemId')[0].getSelection();
20        console.log(records);
21        if(records.length > 0){
22          this.contactListEvent(null,null,null,records[0]);
23        }else{
24          this.getContactDetailView().setActiveItem(0);
25        }
26      }
27
28      Ext.getStore('ContactListStore').clearFilter();
29      var records=this.getContactView().query('#listContactItemId')[0].getSelection();
30      if(records.length > 0){
31        this.contactListEvent(null,null,null,records[0]);
32      }
33    }
34  }
35
36
37
38
39
  
```

- Always code for readability.

```
if (!this.isReadable()) {
    this.refactorWith({
        properIndentation: true,
        optimizedCodeForReadability: true
    });

    else {
        this.beHappy();
    }
}
```

#### 4.4.13 One class per file

- Avoid files with more than 1000 lines of code.
  - Difficult to maintain

```
sendTwitter();
return true;
};

},
),
// eachTime function . Called whenever the plugin is

function($, jQuery, dmJQuery, callback) {
    if (callback) {
        callback();
    }
}
);
```

- Organize your file system.
- Files and folders should match namespacing.
- Follow architectural pattern (MVC or MVVM).
- Abstraction!
- Development loader / Production builder.

#### 4.4.14 Too much work to return

- Make it easy!
- **This code is not wrong but could be better**

```
testSomeVal: function(someVal) {
    if (someVal <= 2) {
        return true;
    } else {
        return false;
    }
}
```

- **We are starting to get there.**

```
testSomeVal: function(someVal) {
    return (someVal <= 2) ? true : false;
}
```

- **Yup, this is it !**

```
testSomeVal: function(someVal) {
    return someVal <= 2; // May be hard to read at first glance.
}

testSomeVal: function(someVal) {
    return (someVal <= 2); // Add braces for readability
}
```

#### 4.4.15 Comments or Documentation

- Bad practice
  - No comments
  - Variables with unmeaningful names
  - Impossible to figure out what is the intent of this code

```
var bs_note = 1;
var arr_p = new Array();
for (var j=0; j<6; j++)
{
    if (!isNaN(tmp[j] && bs_note)
    {
        arr_p[j] = '<div class=\"oo-str\"></div>';
        bs_note = 8;
    }
    else if (!bs_note && isNaN(tmp[j]) && tmp[j] != 'x')
    {
        arr_p[j] = '<div class=\"b-str\"></div>';
    }
    else if (tmp[j] == 'x')
    {
        arr_p[j] = '<div class=\"x-str\"></div>';
    }
    else
    {
        arr_p[j] = '<div class=\"o-str\"></div>';
    }
}
```

- Best practice
  - Comment top-level structures.
  - Use meaningful names : "Self-commenting" code.
  - Add notes whenever logic is not obvious.
- Build your docs into a searchable tool.
  - JSDuck – <https://github.com/senchalabs/jsduck/wiki>
  - API documentation generator for Sencha.



#### 4.4.16 Operators

Use "===" "!===" instead of "==" "!="

- Comparator operations
  - When you receive data in a json structure and you are not sure about what you are getting you should be more restrictive.

#### Truthy and Falsy

When using "==" or "!=", JavaScript has room to coerce values

Tests are boiled down to "Falsy" and "Truthy"

• Process is called coercion

```
console.log('' == '0');           // false
console.log(0 == '0');           // true
console.log(false == 'false');   // false
console.log(false == '0');       // true
console.log(false == undefined); // false
console.log(false == null);      // false
console.log(null == undefined); // true
```

More predictable workflow when program is running

Is going to guarantee the result you are looking for

"===" || "!=="

Leaves no room for coercion

• All are false!

```
console.log('' === '0');          // false
console.log(0 === '0');           // false
console.log(false === 'false');   // false
console.log(false === '0');       // false
console.log(false === undefined); // false
console.log(false === null);      // false
console.log(null === undefined); // false
```

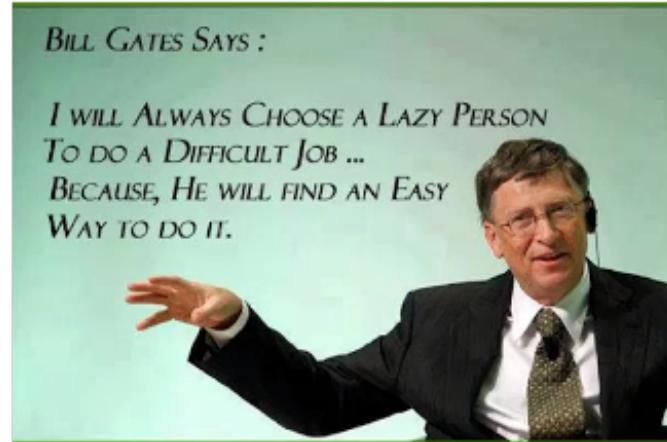
#### 4.4.17 Be lazy

- Bad practice: Initialization of all three panels

```
{
  xtype: 'container',
  layout: 'card',
  items: [
    {
      xtype: 'panel',
      title: 'Card One'
    },
    {
      xtype: 'panel',
      title: 'Card Two'
    },
    {
      xtype: 'panel',
      title: 'Card Three'
    }
  ]
}
```

- Best practice

- Lazy initialization: Add items/views only when necessary
- Lazy rendering: Save the browser some time!
- Reuse things: Save yourself some time!



#### 4.4.18 Knowing this

- This
  - Describes the actual object application is executing
  - Defines context and scope.
- Two rules for this:
  1. When a function is executed via a **var** reference, the default execution context ("this") is **window**
  2. When a function is executed via an object key, the execution context ("this") is the object.

```
var myFn = function() {  
    console.log(this)  
};  
myFn();
```

```
var person = {  
    name: 'jay',  
    getName: function() {  
        console.log(this);  
    }  
};  
person.getName();
```

#### 4.4.19 Additional resources

- "Maintainable JavaScript" by Nicholas Zacha
  - <http://www.slideshare.net/nzakas/maintainable-javascript-2012>
- "Code Conventions for JS" by Douglass Crockford
  - <http://javascript.crockford.com/code.html>
- "JavaScript Performance Tips & Tricks" by Grgur Grisogono
  - <http://moduscreate.com/javascript-performance-tips-tricks/>

## 4.5 Devon4Sencha Tools

### 4.5.1 Code Analysis Tools

- There are a couple of tools you can use to analyse your code. Popular tools for JavaScript development are:
  1. JSLint – <http://www.jslint.com/>
    - A JavaScript syntax checker and validator on coding rules.
  2. JSHint – <http://jshint.com/>
    - A community driven fork of JSLint, which is not as strict as JSLint.
- There are many plugins available for IDE's and editors to check the JavaScript code while writing using the above tools.

### 4.5.2 Analysing code with Sencha Cmd

- Every time when you run a sencha app build or sencha app build testing on the command-line, it will validate your JavaScript code. Lint errors will show up as parse warnings in your console.

#### Using devcon

You can also use the build option using devcon with the devon sencha build command [learn more here](#)

- Not only it checks your JavaScript errors, it will also check your Sass stylesheet for errors, before compiling it to production ready CSS.

```
-----  
[INF] Appending content to F:\xampp\htdocs\liuda\bootstrap.json  
[WRN] C1014: callParent has no target <this.callParent in Ext.Decorator.setDisabled> -- F:\xampp\htdocs\liuda\touch\src\Decorator.js:158  
[WRN] C1014: callParent has no target <this.callParent in Ext.data.ArrayStore.loadData> -- F:\xampp\htdocs\liuda\touch\src\data\ArrayStore.js:65  
[WRN] C1014: callParent has no target <me.callParent in Ext.dataview.DataView.onAfterRender> -- F:\xampp\htdocs\liuda\touch\src\dataview\DataView.js:893  
[WRN] C1014: callParent has no target <this.callParent in Ext.fx.animation.Wipe.getData> -- F:\xampp\htdocs\liuda\touch\src\fx\animation\Wipe.js:120:7  
[WRN] C1014: callParent has no target <this.callParent in Ext.slider.Toggle.setValue> -- F:\xampp\htdocs\liuda\touch\src\slider\Toggle.js:64  
[INF] Concatenating output to file F:\xampp\htdocs\liuda\build\production\liuda\
```

### 4.5.3 Sonar for JavaScript

- The JavaScript plugin enables analysis and reporting on JavaScript projects.
- More information:
  - <http://docs.sonarqube.org/display/PLUG/JavaScript+Plugin>

# sonarqube

Evolución en el tiempo...

Severidades	Reglas
Bloqueante 0	Lines should not end with trailing whitespaces 520
Critica 0	Sections of code should not be "commented out" 95
Mayor 806	"" and "!=" should be used instead of "==" and "!=" 74
Menor 107	Unused function parameters should be removed 26
Info 2	Functions should not be too complex 20
	Collapsible "if" statements should be merged 13

src/main/webapp/js 231 SuperBoxSelect.js 197

src/main/webapp/js/grid 215 SuperBoxSelect.js 197

src/main/webapp/js/superboxselect 197 LockingGridView.js 81

src/main/webapp/js/FileUpload 163 Ext.ux.FileUploader.js 63

Ext.ux.UploadPanel.js 55

SonarQube™ technology is powered by SonarSource SA  
Version 4.5.5 - [LGPL v3](#) - [Community](#) - [Documentation](#) - [Get Support](#) - [Plugins](#) - [Web Service API](#)

## 4.6 How to do effective Devon4Sencha code reviews

### 4.6.1 Benefits

- Gain validation of your approach
- Increase Team Skills
- Improve code quality
- Increase code maintainability
- Improve code testability
- Find bugs

### 4.6.2 Best practices

#### 1. Architecture / Design

- Single responsibility principle : A class should have one-and-only-one responsibility.
- Code duplication.
- Code left in a better state than found.
- Potential bugs.
- Error handling :Are errors handled gracefully and explicitly where necessary?
- Efficiency

#### 2. Style

- Method names
- Variable names
- Function length
- Class length
- File length
- Docstrings : For complex methods, is there a docstring explaining them?
- Commented code : Good idea to remove any commented out lines.
- Number of method arguments : Do they have 3 or fewer arguments?
- Readability : Is the code easy to understand?

#### 3. Testing

- Test coverage : Are the tests thoughtful? Do they cover the failure conditions? Are they easy to read?
- Testing at the right level : Are we as low a level as we need to be to check the expected functionality?

- Meet requirements :
  - Usually as part of the end of a review
  - Take a look at the requirements of the story, task or bug.

#### 4. Review your own code first

- Did I leave a comment or TODO in?
- Does that variable name make sense?
- ... and anything else that we have seen above.

#### 4.6.3 How to handle code reviews

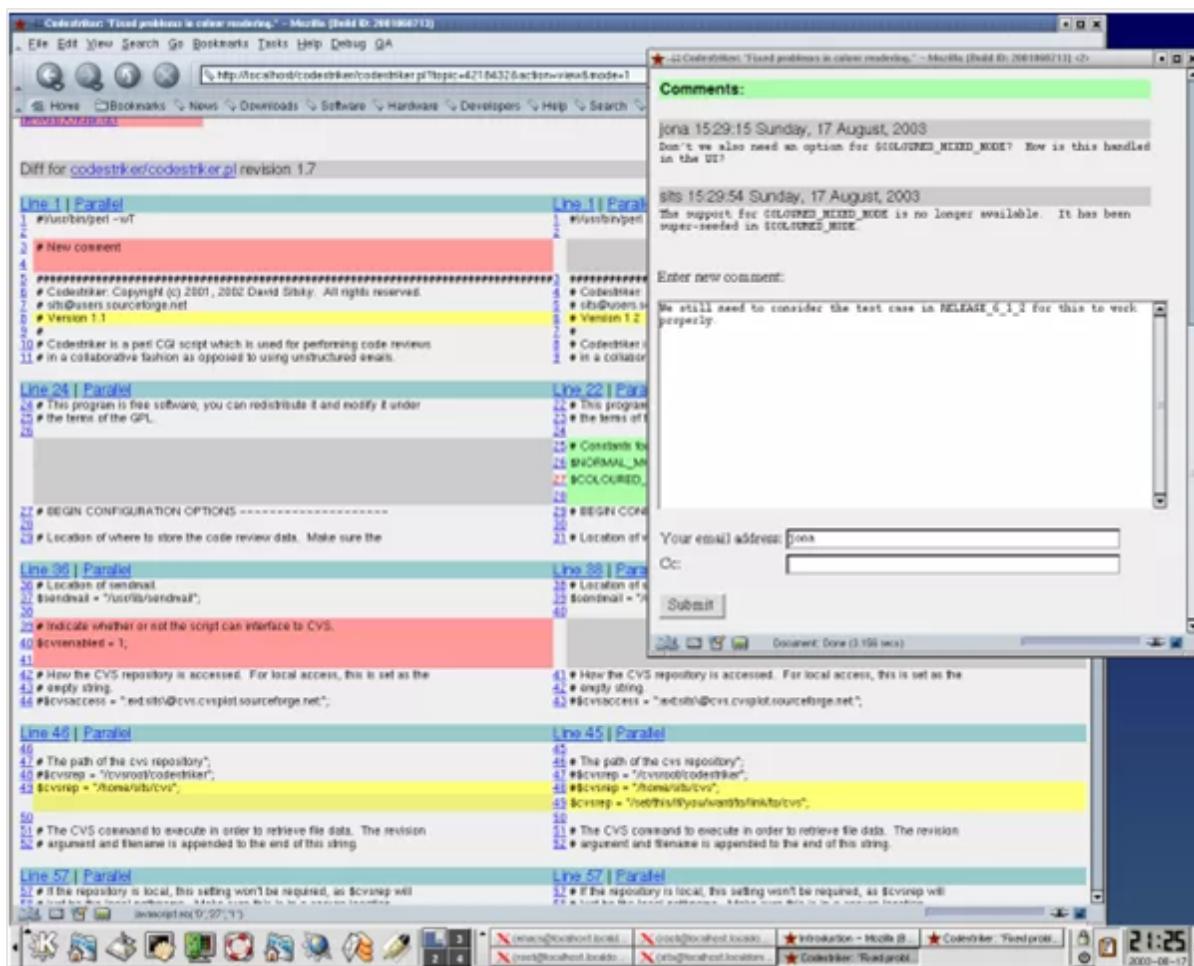
- Ask questions
  - How does this method work?
- Compliment/reinforce good practices
  - Reward developers for growth and effort
- Discuss in person for more detailed points
- Explain reasoning
  - Ask if there's a better alternative and justify why
- Make it about the code
  - Make discussions about the code, not about the developer. It's about improving the quality of the code.
- Suggest importance of fixes
  - It makes the results of a review clear and actionable.

#### 4.6.4 On mindset

- As developers, we are responsible for making both working and maintainable code.
- Improving the maintainability of the code can be just as important as fixing the line of code that caused the bug.
- Keep an open mind during code reviews.

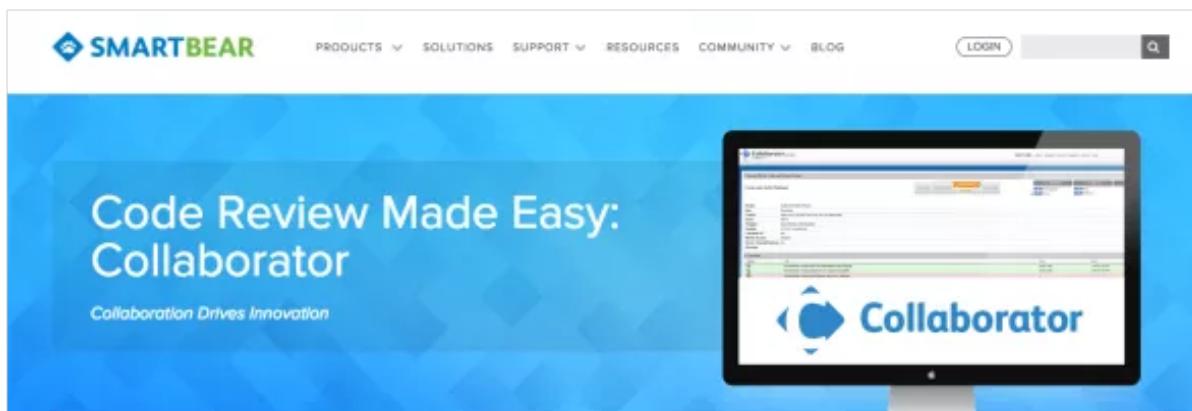
#### 4.6.5 Code review tools

- **Codestriker**
  - Free & open source web application that help developer to web based code reviewing.
  - Developers can ensure issues, comments and decisions are recorded in a database, and provides a comfortable workspace for actually performing code inspections.



- **Collaborator**

- Code review tool that helps development, testing and management teams work together to produce high quality code.
- It allows teams to peer review code, user stories and test plans in a transparent, collaborative framework instantly keeping the entire team up to speed on changes made to the code.



## 4.7 Devon4Sencha testing tools

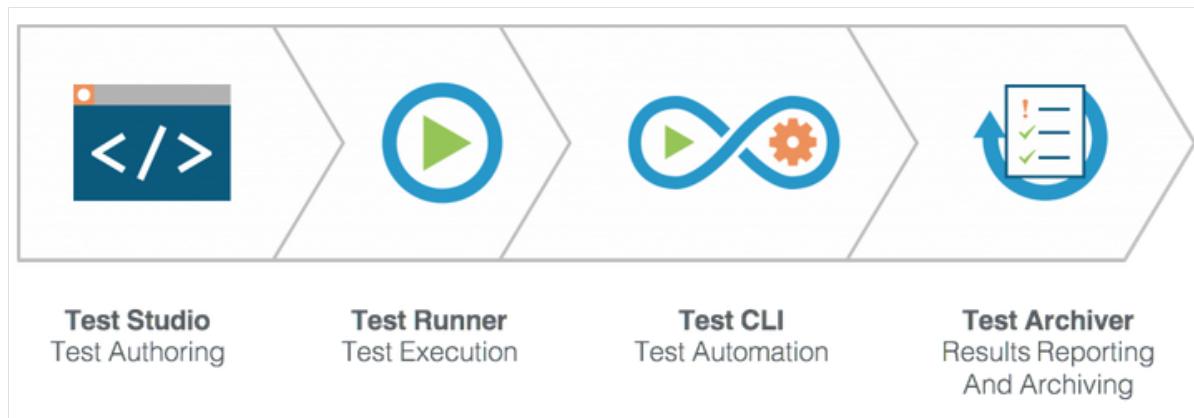
### 4.7.1 Tools for testing your Sencha code

- **Jasmine** – <http://jasmine.github.io/>
  - Open source unit testing framework for JavaScript
  - Unit Test attempt to isolate small pieces of code and objectively verify application logic.
  - Aims to run on any JavaScript-enabled platform, to not intrude on the application nor the IDE, and to have easy-to-read syntax.
  - See also: <https://vimeo.com/18100173>
- **Siesta** – <http://www.bryntum.com/products/siesta>
  - Test any JavaScript code and also perform testing of the DOM and simulate user interactions.
  - UI test attempt to subjectively verify that elements on the screen behave (and often look) as expected statically and dynamically.
  - Automatic tests
  - See also
    - 1. <https://saucelabs.com/> and
    - 2. <http://www.browserstack.com/>
- **Sahi** - <http://sahipro.com/>
  - Works on Every browser.
  - Stable & Easy to maintain automation.
  - Framework & Technology agnostic.
  - Fast Parallel batch playback
  - AJAX timeout? Not with Sahi
  - Impressive Reports & Logs
  - Simple powerful scripting
  - Inbuilt Excel framework
  - Smart accessor identification
- **HP UFT**

### 4.7.2 Sencha Test

- **Sencha Test** <https://www.sencha.com/blog/sencha-test-is-now-generally-available>
- Helps existing and new Ext JS developers to release quality applications.

- Perform cross-browser and cross-platform testing using one comprehensive solution.
- Significantly reduce your software release cycles.
- Sencha Test integrates some of the leading open source test frameworks, including Jasmine and Istanbul.
- Sencha Test components:



- **Sencha Test** – <https://www.sencha.com/blog/sencha-test-is-now-generally-available>

- Supported browsers and Platforms.

Browser Desktop (PCs & Laptops)	Browser Mobile (Tablets & Smartphones)	Test Framework Integrations	Supported Sencha Applications	Test Automation Integrations
Internet Explorer 8+	IE 10+ on Windows Phone 8+	Jasmine 2.4.1	Ext JS 4.2+	Browser Farm – Sauce Labs+
Microsoft Edge	Chrome / Stock Browser on Android 4+	Istanbul 0.4.1	Sencha Touch 2.0+	Continuous Integration – Team City, Jenkins
Chrome	Safari iOS 6+			
Firefox				
Safari 6+				
Opera 15+				

## 4.8 Adapting devon4sencha apps to microservices

### 4.8.1 Introduction

In order to use *devon4sencha* applications with the [Devonfw implementation for microservices](#), we will need to adapt the Security configuration of the framework to authenticate our client app against the *auth* service of the microservices solution.

### 4.8.2 Security changes

To do so we will need to add some changes in the way *Sencha* authenticates. The changes only affect to the `devon4sencha\packages\local\devon-ext.js\src\Security.js` file so we can open it with an editor and add bellow code:

- changing the *loginOperation*: In microservices we can not use the *session* as a security resource so we will start using *security tokens* to allow our apps to authenticate. During logging process, when it is successful, we are going to check if the server is based on microservices (if headers *accessHeaderName* and *accessToken* are presents) and in this case we are going to store the obtained *token* in a security variable, and schedule a refresh operation for the token. The new *loginOperation* function will look like bellow

```
loginOperation: function(options) {
    var me = this;

    Devon.rest.security.login.post({
        jsonData: {
            j_username: options.user,
            j_password: options.password
        },
        success: function(data, response, opts) {

            //Check if JWT security is activated
            if (data && data.accessHeaderName && data.accessToken) {
                me.jwtData = data;
                var headers = {};
                headers[data.accessHeaderName] = data.accessToken;
                Ext.Ajax.setDefaultHeaders(headers);
                if (options.success) {
                    options.success.call(options.scope, this.currentUser);
                }
            }

            var task = {
                run: function(){
                    me.refreshJwtData();
                    return true;
                },
                interval: data.expirationTime / 3,
                scope: me
            }
            Ext.TaskManager.start(task);
        }

        options.success(data, response, opts);
    },
    failure: options.failure,
    scope: options.scope
});
}
```

- Adding the refresh token function: The *refreshJwtData* function will be

```
refreshJwtData: function(options) {
```

```
var me = this;

var headers = {};
headers[me.jwtData.accessHeaderName] = me.jwtData.accessToken;
headers[me.jwtData.refreshHeaderName] = me.jwtData.refreshToken;

Devon.rest.security.jwtRefresh.post({
headers: headers,
success: function(data, response, opts) {
if (data && data.accessHeaderName && data.accessToken) {
me.jwtData = data;
var headers = {};
headers[data.accessHeaderName] = data.accessToken;
Ext.Ajax.setDefaultHeaders(headers);
}
}
});
}
```

- changing `getCSRF` function: we will check if the `token` related to microservices is present. If that is the case we are going to call directly the `options.success.call`

```
getCSRF: function(options) {
Devon.Log.trace("->onUserLoaded");

//JWT Security
if (this.jwtData != null) {
options.success.call(options.scope, this.currentUser);
}

//JsessionId Security
else {
Devon.rest.security.csrftoken.get({
scope: this,
failure: Ext.emptyFn,
success: function(csrf, response, opts) {
if (csrf && csrf.headerName && csrf.token) {
this.csrf = csrf;
var headers = {};
headers[csrf.headerName] = csrf.token;
Ext.Ajax.setDefaultHeaders(headers);
if (options.success) {
options.success.call(options.scope, this.currentUser);
}
}
},
callback: function() {
if (!this.csrf) {
if (options.failure) {
options.failure.call(options.scope);
}
}
}
});
}
}
```

With above changes the `devon4sencha` app will be able to authenticate against a `devonfw microservices` app.

- Versions used to create this guide

Software	version
devonfw	2.0.1
Sencha Cmd	6.2.1.25
NodeJS	4.5.0
Cordova	6.4.0
Android SDK	24.4.1

## 4.9 How to start a cordova project from a sencha project

### 4.9.1 Recommended readings:

- [Integrating With Cordova or PhoneGap](#)
- [Introduction to Cordova](#)

### 4.9.2 Minimum requirements:

- Downloaded and deployed [devonfw](#)
- NodeJs installed (using the one included in devonfw or another one)
- Java JDK 7 or superior (required by Cordova)

#### Optional requirements based on the platform to deploy

- **Android:** Android SDK, needed to create the .apk file and deploy it on the Android device/emulator. [Android Platform Guide by Cordova](#)
- **Windows Phone:** Windows 8.1 or Windows 10 to be able to install the required Visual Studio and the required tools. [Windows Platform Guide by Cordova](#)
- **iOS:** Mac computer, to be able to open the project in XCode, create the .ipa file and deploy it on a iOS device/emulator. [iOS Platform Guide by Cordova](#)

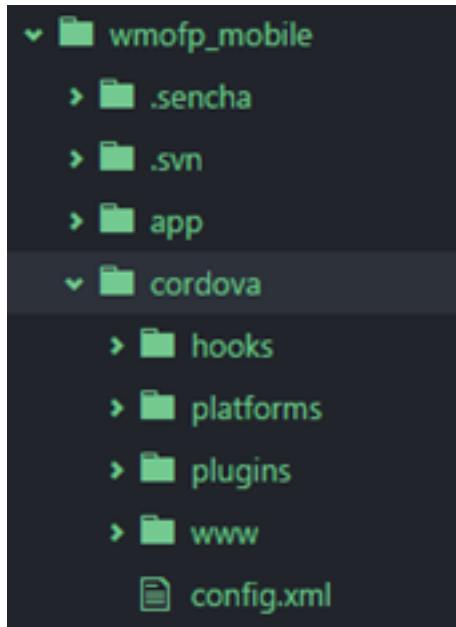
### 4.9.3 Steps to do on the Sencha project

#### Initialize Cordova

Open the devon console, navigate to the project folder and then run the following command:

```
sencha cordova init com.mycompany.MyApp MyApp
```

Once the command finishes, the folder "cordova" must appear on the structure of the project.



## Modifying the app.json

To indicate to sencha that we have several platforms, and copy the files needed inside the "cordova" folder, we need modify the app.json on the Sencha project to add the required platforms/configurations.

As follows you can see an example of a project configured to run on iOS, Android and Windows Phone (8.X or superior).

### app.json.

```
"builds": {  
  "modern": {  
    "toolkit": "modern"  
  },  
  "android": {  
    "toolkit": "modern",  
    "packager": "cordova",  
    "cordova": {  
      "config": {  
        "platforms": "android",  
        "id": "com.mycompany.MyApp"  
      }  
    }  
  },  
  "ios": {  
    "toolkit": "modern",  
    "packager": "cordova",  
    "cordova": {  
      "config": {  
        "platforms": "ios",  
        "id": "com.mycompany.MyApp"  
      }  
    }  
  },  
  "windows": {  
    "toolkit": "modern",  
    "packager": "cordova",  
    "cordova": {  
      "config": {  
        "platforms": "windows",  
        "id": "com.mycompany.MyApp"  
      }  
    }  
  }  
},
```

```
    }
}
```

**Note**

Notice that the name of the build is not necessary the name of the platform to deploy, could be a different one. As developers we can create many configurations as we need, configuration for a specific device, to try different themes, etc.

#### 4.9.4 Steps to do on the Cordova project

##### Adding platforms

Open a console (must have the path of Cordova executable in the PATH variable, could be the Devon console or not), then proceed to add all the platforms wanted ([Add Platforms by Cordova](#))

Example of a project where add Android, iOS and Windows Phone

```
cordova platform add android
```

**Note**

If the Android SDK is missing, an error will be returned. "Error: Failed to find 'ANDROID\_HOME' environment variable...."

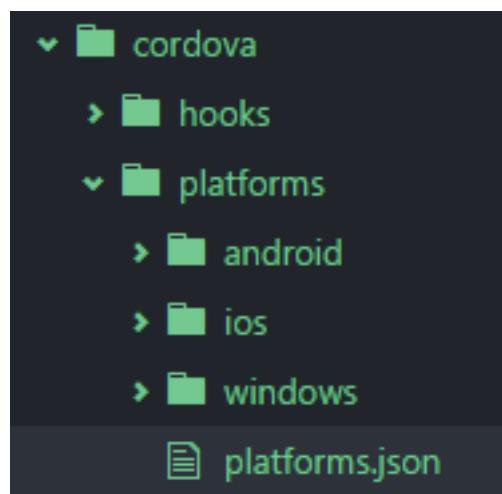
```
cordova platform add ios
```

**Note**

If you run this command in a non Mac OS, a warning will appear indicating that the packing and deployment of the app will be not possible

```
cordova platform add windows
```

Once all the platforms are added, the Cordova project will appear as follows:



**platform.json.**

```
{  
  "android": "6.1.0",  
  "ios": "4.3.0",  
  "windows": "4.4.3"  
}
```

- Can check the latest version of Android plugin [here](#)
- Can check the latest version of iOS plugin [here](#)
- Can check the latest version of windows plugin [here](#)

#### 4.9.5 Steps to run the Sencha project in the Cordova project

##### Sencha app build

We can use several ways to deploy the Sencha application in the cordova folder, to have all the files updated and ready to see the application in the desired device.

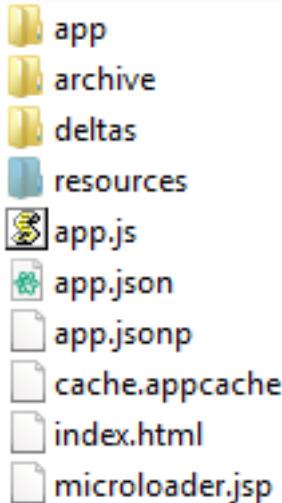
##### Do a regular build, copy the files manually and run cordova

This is the most basic way to have all the required files to run our application in Cordova. To see our progress in Sencha, usually we use the *watch* command. For this scenario we need to use the *build* one, this option is more restrictive than the *watch*, and creates a ZIP file with all the files required for the webapp.

```
sencha app build [modern]
```

##### Note

If the app.json has several build config, sencha will try to do a build for each one. Bear this in mind to avoid several builds and run just the desired. You can specify the configuration for the build adding the name of the configuration at the end.



Once the previous command is finished, open the build ZIP file and replace all the resources inside the folder "myApp/cordova/www", then you have all the files updated in the cordova folder.

Once we have all the files updated, the next step is to do a "prepare/build/run" in Cordova for the desired platform. To see more information, click [here](#)

```
cordova prepare android
cordova build ios
cordova run windows
```

**Note**

**PREPARE:** Transforms config.xml metadata to platform-specific manifest files, copies icons & splashscreens, copies plugin files for specified platforms so that the project is ready to build with each native SDK.

**Note**

**BUILD:** Shortcut for "cordova prepare" + "cordova compile".

**Note**

**RUN:** Prepares, builds, and deploys the app on specified platform(s) devices/emulators.

**Use a specific build config**

As part of the changes mentioned in this document, we modify the app.json file to add some additional builds. This build configurations make able to "preapre/build/run" our Sencha application inside the cordova folder for a specific platform. Instead of doing a generic build, we are going to take advantage of that specific configurations.

```
sencha app prepare android
sencha app build ios
sencha app run windows
```

**Note**

**PREPARE:** Transforms config.xml metadata to platform-specific manifest files, copies icons & splashscreens, copies plugin files for specified platforms so that the project is ready to build with each native SDK.

**Note**

**BUILD:** Shortcut for "cordova prepare" + "cordova compile".

**Note**

**RUN:** Prepares, builds, and deploys app on specified platform(s) devices/emulators.

[[devonfwguide\_the-presentation\_layer: getting started with oasp4js]] == The Presentation Layer: Getting started with OASP4JS

## 4.9.6 OASP4Js

Created to support the development of Angular applications, [OASP4Js](#) includes Google Material Design as main visual language to take at maximum the advantage of Angular possibilities and Material components, making possible the building of modular, well-designed, responsive front-end applications.

[[getting-started-oasp4js-introduction\_oasp4js-technology stack]] === Oasp4Js Technology stack

OASP4Js works on top of Angular but also, it provides several tools, libraries and code conventions to make your Angular apps easier to develop based on the following technologies:

- [Angular Framework](#) as the main development framework
- [Google Material2](#) as visual language and components.
- [Covalent Teradata](#) as Component and utilities library working with Google Material.
- [Yarn](#) as Project dependencies management tool.

The main advantages of this technologies are:

- Teradata provides:
  - 4 available layouts that fits nowadays design necessities.
  - Several tools and utilities regarding style conventions such as text size, padding, margins...
  - Complex components as: Data tables, Chips with autocomplete, pagination...
- Google Material component library is composed by a number of fancy components like tabs, cards, buttons...
- Yarn is quite faster than NPM and provides some more functionalities to manage dependencies.

## Oasp4Js tools

### IDE

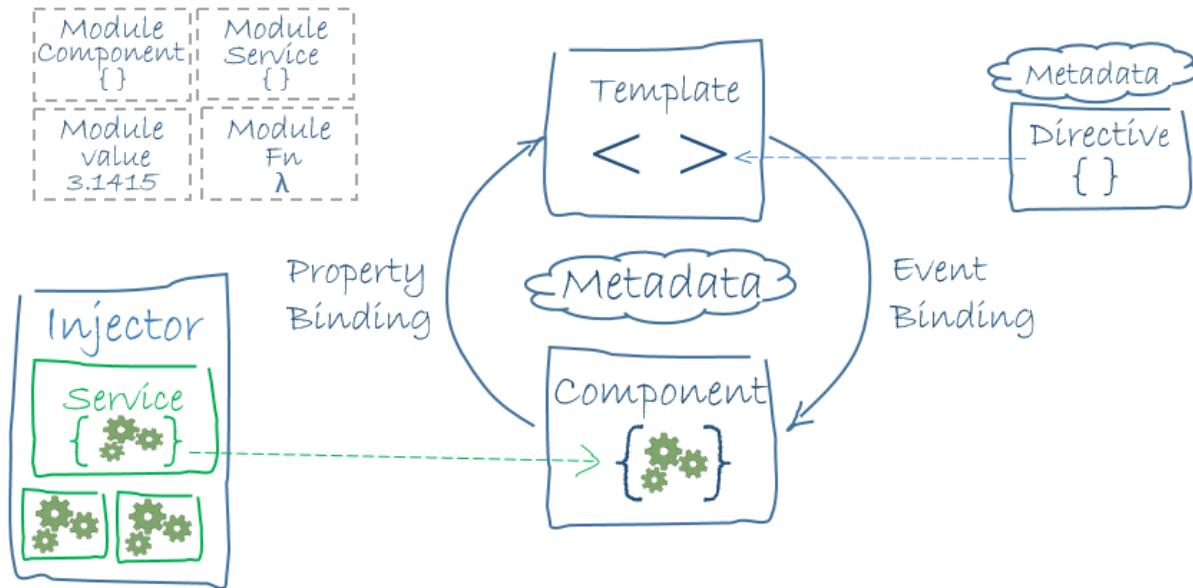
There is no integrated IDE with the framework, that means you are free to work with whatever IDE fits better with you. Event though, we fully recommend the use of [Visual Studio Code](#), along with a [Guide of the most interesting plugins](#) to make even easier your development with Typescript and Angular.

[[getting-started-oasp4js-introduction\_angular/cli]] ===== Angular/Cli This [CLI](#) helps the developer to automatize some common processes. It comes with [Webpack](#) as main bundler. It is widely used in the Angular community thanks to the boost of productivity it provides at the time of creating new projects from scratch, serving and testing the project, creating new components, services, directives...

### Testing

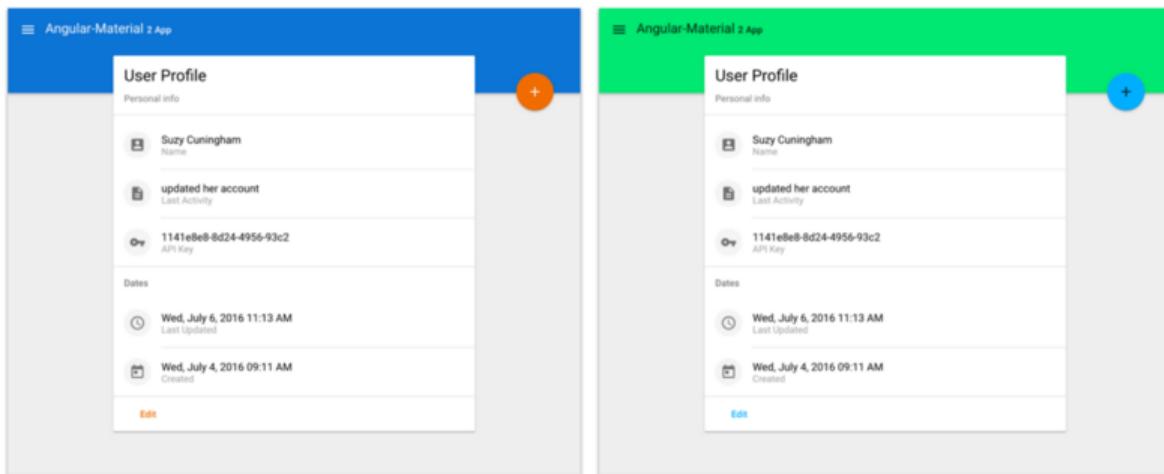
Testing helps the developers to be sure that all the typescript in services and components works properly, but also, it can test HTML tags and properties. There are many options to test an Angular app, the default option is [Karma](#) and [Jasmine](#).

[[getting-started-oasp4js-introduction\_oasp4js-architecture\_overview]] === Oasp4Js architecture overview As Angular is the main framework for Oasp4Js, the architecture of the applications that is going to be used is the same as [Angular](#):

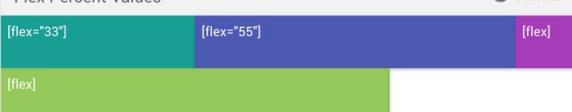
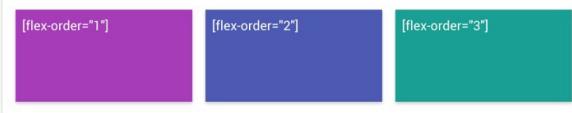


This architecture will be enhanced with some functionalities from Covalent Teradata and Angular Material:

- Theming: functionality that Angular Material includes in its library and Covalent Teradata extends, it declares one primary color, one secondary color and one color for warning and alerts to be used in all the application. Also Covalent Teradata expects a color for the background and another for the foreground. These colors will be stored in one theme, where you can store as much as you want and be changed at the run-time by the user.



- Flex-box: Along with other [CSS Utility Styles & Classes](#), Covalent Teradata comes with flex-box, useful for styling and organizing components inside of a view, which also has been extended by Covalent Teradata to achieve responsiveness. You can declare styles that change, hide or transform the component depending on the screen resolution of the device.

<p>Flex Attribute</p>  <p>[flex]</p> <p>[flex]</p> <p>[flex]</p>	<p>layout="row" layout-align="space-around center"</p>  <p>one</p> <p>two</p> <p>three</p>
<p>Flex Percent Values</p>  <p>[flex="33"]</p> <p>[flex="55"]</p> <p>[flex]</p>	<p>layout="row" layout-align="space-between center"</p>  <p>one</p> <p>two</p> <p>three</p>
<p>Flex Order Attribute</p>  <p>[flex-order="1"]</p> <p>[flex-order="2"]</p> <p>[flex-order="3"]</p>	<p>Responsive Flex &amp; Offset Attributes</p> <p>I flex to one-third of the space on mobile, and two-thirds on other devices.</p> <p>I flex to two-thirds of the space on mobile, and one-third on other devices.</p>

In the following chapters we are going to see an example of a finished Oasp4Js application, how to build your own Oasp4Js app from the scratch and all you need to know about about each part of the architecture in order to start building your own applications.

#### 4.9.7 Oasp4js tutorial

You can follow the [Oasp4js tutorial](#): a complete tutorial of a client implementation based on *Oasp4j* ([Angular](#)). In this tutorial you will see how to implement a basic (but fully functional) client application based on *Oasp4js*, from how to start new applications to how to connect with a backend service.

## 5. Appendix

### 5.1 Frequently Asked Questions (FAQ)

#### 5.1.1 Server

##### Security

[[devon-guide-frequently-asked-questions\_how-to disable spring boot security?]] ===== How to disable Spring Boot Security?

[[devon-guide-frequently-asked-questions\_disable-security for the services]] ===== Disable security for the services

In case, you want to disable the default security filter of roles, to have full access to the services of the sample application, you must introduce the following changes:

[[devon-guide-frequently-asked-questions\_modify-the unsecuredresources]] ===== Modify the *unsecuredResources*

In the *BaseWebSecurityConfig.java* class located in the *src/main/java/io.oasp.gastronomy.restaurant/general/service/impl/config/BaseWebSecurityConfig.java* folder of the *core* project, add the rest services path to the *unsecuredResources*

```
String[] unsecuredResources =  
    new String[] { "/login", "/security/**", "/services/rest/login", "/services/rest/logout", "/  
services/rest/**" };
```

[[devon-guide-frequently-asked-questions\_disable-csrf protection]] ===== Disable CSRF protection

Also, in *BaseWebSecurityConfig.java*

- comment the *CsrfRequestMatcher*

```
// .csrf().requireCsrfProtectionMatcher(new CsrfRequestMatcher()).and()
```

- add right after

```
.csrf().disable()
```

In the *CsrfRequestMatcher.java* class in *'src/main/java/io.oasp.gastronomy.restaurant/general/common/impl/security'*, change *\_PATH\_PREFIXES\_WITHOUT\_CSRF\_PROTECTION* to

```
private static final String[] PATH_PREFIXES_WITHOUT_CSRF_PROTECTION = { "/login", "/logout", "/services/  
rest/**", "/websocket" };
```

[[devon-guide-frequently-asked-questions\_disable-the global method security]] ===== Disable the Global Method Security In the *SpringBootApp.java* class, located in *src/main/java/io.oasp.gastronomy.restaurant*, change (to *false*) the *EnableGlobalMethodSecurity* annotation.

```
@EnableGlobalMethodSecurity(jsr250Enabled = false)
```

[[devon-guide-frequently-asked-questions\_check-the services access]] ===== Check the services access Now, run the app (Right click over *SpringBootApp.java* class and select *Run As > Java Application*) and try to access the rest service through a browser:

```
http://localhost:8081/oasp4j-sample-server/services/rest/tablemanagement/v1/table/101
```

Should return a response similar to

```
{"id":101,"modificationCounter":1,"revision":null,"waiterId":null,"number":1,"state":"OCCUPIED"}
```

### Disable Login

In addition to the previous section, if you also want to disable the security login of the sample application, you only need to delete/comment the content of the *configure* method of the *BaseWebSecurityConfig.java* class.

```
@Override  
public void configure(HttpSecurity http) throws Exception {  
}
```

Doing that, you will directly access the *Welcome* page of the app without the login process.

[[devon-guide-frequently-asked-questions\_\_load-denied by x-frame error\_ when trying to upload a file to the server from the client]] ===== *Load denied by X-Frame error* when trying to upload a file to the server from the client

This error is thrown by the spring security component in the server in order to avoid [clickjacking](#) attacks. You can override this configuration in the spring security config adding the following:

```
---  
<http>  
  <headers>  
    <frame-options policy="SAMEORIGIN"/>  
  </headers>  
</http>  
---
```

The available options for *policy* are: *DENY*, *SAMEORIGIN* and *ALLOW-FROM*. You can get more information about this, [here](#) and [here](#)

## 5.1.2 Client

### Devon4Sencha

[[devon-guide-frequently-asked-questions\_\_how-to disable defeat cache functionality?]] ===== How to disable *defeat cache* functionality? When working with Sencha application, all the required files are loaded with an additional parameter *\_dc=[random number]* that will disallow debugging properly on the browser developer tools because breakpoints will be lost on reload.

In order to disable this functionality, you have to add the following on the *app.json* of your Devon4Sencha application:

```
  "loader": {  
    "cache": true  
  },
```

### 5.1.3 Configuration issues

#### Maven

##### Note

Remember to verify the proxy configuration in the `conf/.m2/settings.xml` file and the Firewall configuration of your machine as first steps, if you have connection issues using Maven.

[[devon-guide-frequently-asked-questions\_non-resolvable-import pom: project dependencies download failure]] ===== Non-resolvable import POM: Project dependencies download failure

#### Error details

- Could not transfer artifact from/to central ValidatorException

```
[source,batch]
-----
Non-resolvable import POM: Could not transfer artifact org.springframework.boot:spring-
boot-dependencies:pom:1.3.3.RELEASE from/to central (https://repo.maven.apache.org/
maven2): sun.security.validator.ValidatorException: PKIX path building failed:
  sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path
  to requested target
-----
```

- Maven Dependency Problem: Failed to read artifact descriptor / Missing artifact

#### Solution

- Try changing in the `conf\.m2\settings.xml` file of your distribution the URL of the remote repo from `https` to plain `http` : <http://repo.maven.apache.org/maven2>
- As an alternative, you also can include a `<mirror>` tag inside `<mirrors>` with the following structure:

```
<mirror>
  <id>UK</id>
  <name>UK Central</name>
  <url>http://uk.maven.org/maven2</url>
  <mirrorOf>central</mirrorOf>
</mirror>
```

##### Note

If the project is already imported in Eclipse then update the project: Right click on project > Maven > Update Project > check the Force update of Snapshot/Releases checkbox > Ok

[[devon-guide-frequently-asked-questions\_how-to install the devon ip modules?]] ===== How to install the Devon IP Modules?

If you have any problems resolving the dependencies or accessing to the Devon IP modules, you can install them directly in your distribution. To achieve that, you only need to open a distribution console launching the 'console.bat' script and then go to following path `workspaces\examples\devon\modules` and use the `install` command of `Maven`.

```
D:\Devon-dist\workspaces\examples\devon\modules>mvn install
```

```
[...]  
[INFO] - - - - -  
[INFO] Reactor Summary:  
[INFO]  
[INFO] devonfw-modules ..... SUCCESS [ 0.565 s]  
[INFO] devonfw-foo ..... SUCCESS [ 2.969 s]  
[INFO] devonfw-reporting ..... SUCCESS [ 10.022 s]  
[INFO] devonfw-winauth ..... SUCCESS [ 3.069 s]  
[INFO] - - - - -  
[INFO] BUILD SUCCESS  
[INFO] - - - - -  
[INFO] Total time: 17.002 s  
[INFO] Finished at: 2016-11-22T15:39:10+01:00  
[INFO] Final Memory: 15M/40M  
[INFO] - - - - -
```

This command will install the available Devon IP modules of your distribution locally, so you will be able to start using them on your Devon application.

**[[devon-guide-frequently-asked-questions\_address-is invalid on the local machine, or port is not valid on remote machine\_error]]** ===== *Address is invalid on the local machine, or port is not valid on remote machine* error

This error is related mainly to a network problem.

In some environments, Maven seems to be attempting to use an IPv6 address to do the HTTP calls, when either the OS doesn't support it, or is not set up properly to handle it.

You can force Maven (that is a Java tool) to use an IPv4 address with the property.

```
-Djava.net.preferIPv4Stack=true
```

So, in order to use that property, you need to add it to your Maven command:

```
mvn -Djava.net.preferIPv4Stack=true {{lifecycle phase}}
```

## Spring Boot

**[[devon-guide-frequently-asked-questions\_how-to set debug mode for logs?]]** ===== How to set debug mode for logs? For Spring Boot applications, the easiest way is to edit the `[project]\main\resources\application.properties` file and add the following property:

```
logging.level.=DEBUG
```

### Note

The accepted modes for logging level are: TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

**[[devon-guide-frequently-asked-questions\_recommended-usage for annotations]]** ===== Recommended usage for annotations

As a Devon design principle is recommended to use the annotations in *private field* rather than *property access* as it provides a better encapsulation for a similar performance.

### Exceptions

However, you can find an exception to that principle in the case of *Lazy loading in Hibernate*. In this case, using a *field access* to get the *Id* of an Entity Hibernate initializes a lazy proxy that triggers an SQL

query that loads the entire entity from DB, what may cause an important impact over the performance of the application.

So, in this case, instead of using the `@Id` annotation with a private field

```
@Id  
private long id;
```

the alternative might be to use the *property access* using the annotation directly over the *getter*

```
private long id;  
  
@Id  
public String getId() {  
    return this.id;  
}
```

Although, you can also find [other solutions](#) to avoid the *property access*.

You can find more information about this topic here:

- [Yammer discussion](#)
- [Hibernate forum](#)
- [JPA implementation patterns: Field access vs. property access](#)
- [EJB 3.0 Annotations mit Hibernate Lazy Loading](#)
- [Field access vs Property access in JPA](#)

## Tomcat

[[devon-guide-frequently-asked-questions\_\_java.lang.NoSuchMethodError\_error when deploying devon app on tomcat 7]] ===== `java.lang.NoSuchMethodError` error when deploying Devon app on Tomcat 7

### Error details

```
java.lang.NoSuchMethodError: javax.servlet.ServletContext.getVirtualServerName()Ljava/lang/String;
```

### Solution

Add below dependencies in the `pom.xml` file of the core project

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    <exclusions>  
        <exclusion>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-tomcat</artifactId>  
        </exclusion>  
    </exclusions>  
</dependency>  
  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-tomcat</artifactId>  
    <scope>provided</scope>  
</dependency>
```

## 5.1.4 Crosscutting concerns

[[devon-guide-frequently-asked-questions\_which-is the format of a rest request?]] ===== Which is the format of a REST request?

Using Spring Boot and with the default code structure and practices, you can make a GET request to the following url:

```
http://[server]:[port]/services/rest/[service name]/[service version]/[operation]/[param1]
```

For example:

```
http://localhost:8080/services/rest/tablemanagement/v1/table/101
```

Take into account that, for an app deployed to a traditional web server (WAR/EAR packaged), it is usually prepended with the context name of the application, for example:

```
http://oasp-ci.cloudapp.net/oasp4j-sample/services/rest/tablemanagement/v1/table
```

## 5.2 Working with Git and Github

[[devon-guide-working-with-git-and-github\_what-is a version control system]] == What is a version control system

A version control system (VCS) allows you to track the history of a collection of files. It supports creating different versions of this collection. Each version captures a snapshot of the files at a certain point in time and the VCS allows you to switch between these versions. These versions are stored in a specific place, typically called a repository.

[[devon-guide-working-with-git-and-github\_what-are git and github]] == What are Git and Github

Git is currently the most popular implementation of a distributed version control system.

Git originates from the Linux kernel development and was founded in 2005 by Linus Torvalds. Nowadays, it is used by many popular open source projects, e.g., the Android or the Eclipse development teams, as well as many commercial organizations.

The core of Git was originally written in the programming language C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

You can use it on Windows by installing, among others, [Git for Windows](#).

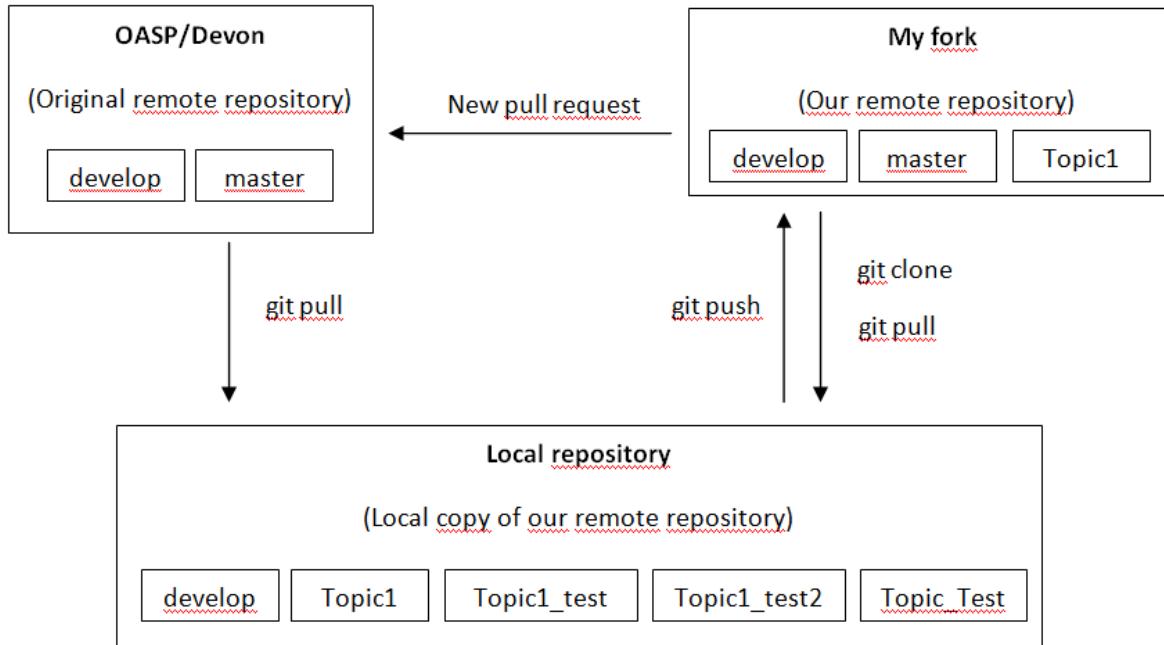
GitHub is a web-based Git repository hosting service. It offers all of the distributed revision control and source code management (SCM) functionality of the Git as well as adding its own features.

Both, the Open Source as well as the IP parts of Devon are hosted on GitHub. The workflow is based on the default workflow as being supported by GitHub.

[[devon-guide-working-with-git-and-github\_devon-and oasp4j workflow for git]] == Devon and OASP4J Workflow for Git

When you work with Git on a [Devon / OASP4J](#) project, you need to take into account that you are always working with a local copy of the remote repository.

The image below can help you to have a clear view about the way to work with Git, on the [Devon / OASP4J](#) project.



[[devon-guide-working-with-git-and-github\_step-1 create a new fork]] === Step 1 Create a new Fork

To avoid problems with OASP/Devon repositories, you need to create your own copy (a "fork") of the repository.

To create the repository, go to the GitHub repository of [Devon](#) / [OASP4J](#) and click on the option 'Fork'.

devonfw / devon PRIVATE

Server reference implementation (integrates IP components on OASP4J)

50 commits 3 branches 0 releases 6 contributors

Branch: develop New pull request New file Upload files Find file HTTPS https://github.com/devonfw/

ivanderk Restored commit: b66b580

modules Restored commit: b66b580 8 days ago

sample Restored 'devonfw' project name and corrected error in pom.xml of dev... a month ago

.gitignore Devon module winauth include .gitignore to root path a month ago

.project JavaDoc include add in winauth module a month ago

README.md Update README.md 11 months ago

pom.xml Version bump -> 2.0.0-SNAPSHOT & simple (silly) documentation of module 29 days ago

README.md

devon

Devon (general and technology-independent IP components and overview)

A fork is a copy of a repository. Forking a repository, allows you to freely experiment with changes without affecting the original project.

Most commonly, forks are used, either to propose changes to someone else's project or to use someone else's project as a starting point for your own ideas.

[[devon-guide-working-with-git-and-github\_step-2 clone the repository]] == Step 2 Clone the repository

Now, copy the forked repo (created in the last step) on your local machine. This process is called "making a clone". To do so, you need to open a console (for example: GitBash) and then execute the next commands, in the directory where you want to create the local copy of the remote repository.

Devon

```
git clone https://github.com/<your_git_user>/devon
```

OASP4J

```
git clone https://github.com/<your_git_user>/oasp4j
```

Now, you have a local copy of the repository.

[[devon-guide-working-with-git-and-github\_step-3 define the repository url]] == Step 3 Define the repository URL

To avoid problems with the Git URLs repositories, you can redefine the label used by git as a shortcut for the repository's URL. The standard label "origin" will be replaced by your GitHub username.

To do so, you need to open the console and go to the local repository and then execute the next commands:

```
git remote add devon https://github.com/devonfw/devon
```

Or

```
git remote add oasp https://github.com/oasp/oasp4j
```

Now, you can see the remote repositories on the command prompt.

```
git remote -v
```

If you are defining Devon URL, you will see something like this:

```
$ git remote -v
devon  https://github.com/devonfw/devon (fetch)
devon  https://github.com/devonfw/devon (push)
origin https://github.com/<your_git_user>/devon (fetch)
origin https://github.com/<your_git_user>/devon (push)
```

If you are adding OASP4j:

```
$ git remote -v
oasp   https://github.com/oasp/oasp4j (fetch)
oasp   https://github.com/oasp/oasp4j (push)
origin https://github.com/<your_git_user>/devon (fetch)
origin https://github.com/<your_git_user>/devon (push)
```

Now, rename the origin remote repository the with following command:

```
git remote rename origin <your_git_user>
```

[[devon-guide-working-with-git-and-github\_step-4 working with topic branches]] === Step 4 Working with Topic Branches

The previous steps were an introduction about how you can get the remote repositories on your local machine. Now, you need to work with this repository. To do so, you need to create a new topic branch.

Topic branches are typically lightweight branches that you create locally and that has a name which is meaningful for you. These topic branches are the one's where you might work, to fix a bug or a feature (they're also called feature branches) that is expected to take some time to complete.

Another type of branch is the "remote branch" or "remote-tracking branch". This type of branch follows the development of someone else's work and is stored in your own repository. You periodically update this branch (using git fetch) to track what is happening there. When you are ready to catch up with everybody else's changes, you would use git pull to both fetch and merge.

To create a new topic branch, you need to use the next command:

```
git branch <new_branch_name>
```

To see the actual branch, you can use the next command:

```
git branch
```

To view all the branches, you can use the following command. Also, you can use this command to view the actual branch as it's with an asterisk mark.

```
git branch -a
```

To move to another branch, you need to use:

```
git checkout <name_of_existing_branch>
```

[[devon-guide-working-with-git-and-github\_step-5 commit the changes]] === Step 5 Commit the changes

When you are working in a branch and you want to change the branch or you just want to save your change in your local repository, you need to commit the changes.

To commit your changes, you need to use the following command:

```
git commit -m "Commit message"
```

After executing the above command, git stores the current contents of the index in a new commit along with a log message from the user describing the changes.

In several cases, you will see a message like this:

```
$ git commit -m "Commit message"
On branch new_branch
Changes not staged for commit:
  deleted:   README.md
  modified:  pom.xml
```

```
Untracked files:  
  New Text Document.txt  
  
no changes added to commit
```

Thus, git shows the changes in the branch and you need to add the file "New Text Document.txt". There are several ways to add a new file in git.

You can add file using the following command:

```
git add <file_name>
```

### Note

You need to be careful, if you have any space in the name of the file. You need to add the name as shown below:

```
git add File\ With\ Spaces.txt
```

Another way to add the files is shown below:

```
git add .
```

This command will add all the untracked files in the local repository, this is a little bit dangerous because in some cases, you don't want to add some files, such as Eclipse configuration files.

In this case, you need a way to exclude or ignore some files. Git has a file called `.gitignore`, where you can put the files to ignore. The content of the file looks as shown below:

```
*.class  
*.classpath  
*.project  
*.iml  
.*  
target/  
jsclient/  
eclipse-target/  
**/src/generated/  
**/tmp/  
  
# Package Files #  
*.jar  
*.war  
*.ear  
  
# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml  
hs_err_pid*
```

Thus, there are many extensions and folders that Git will ignore, if you use the command "git add .".

### Note

Windows doesn't permit us to create a file with the name `.gitignore`, so to create a new `.gitignore` file, you can use the following command:

```
echo "" > .gitignore  
git add .gitignore
```

Then, you can open the file with a text editor and include the filenames which you want to ignore.

Another way to commit without any problems is to commit and add the files at the same time, you can do this with the command:

```
git commit -am "Commit message"
```

You need to keep in count the `.gitignore` file in this case too.

[[devon-guide-working-with-git-and-github\_step-6 push to the remote repository]] === Step 6 Push to the remote repository

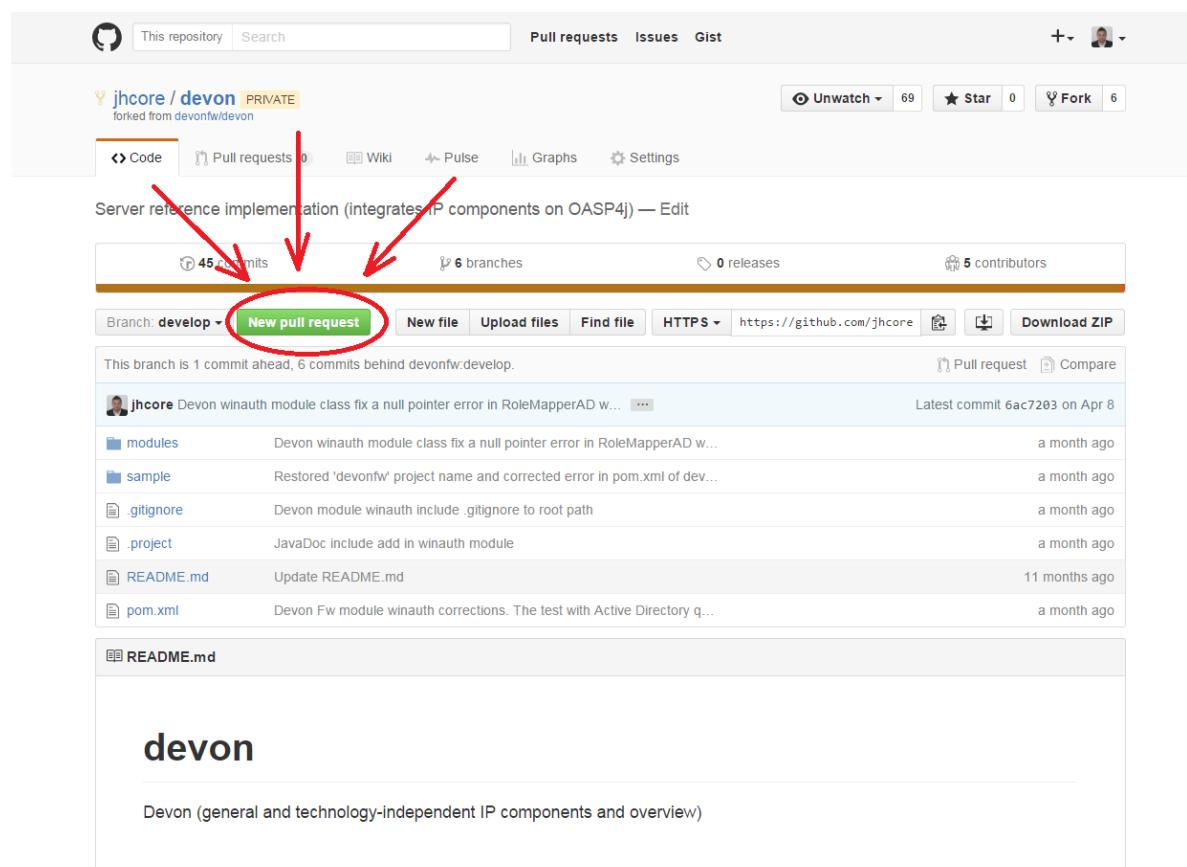
When you want to include your changes in the repository, include it in your remote repository. To do so, you need to push your local topic branch in remote branch.

```
git push <remote_repository> <topic_branch_origin>:<topic_branch_destiny>
```

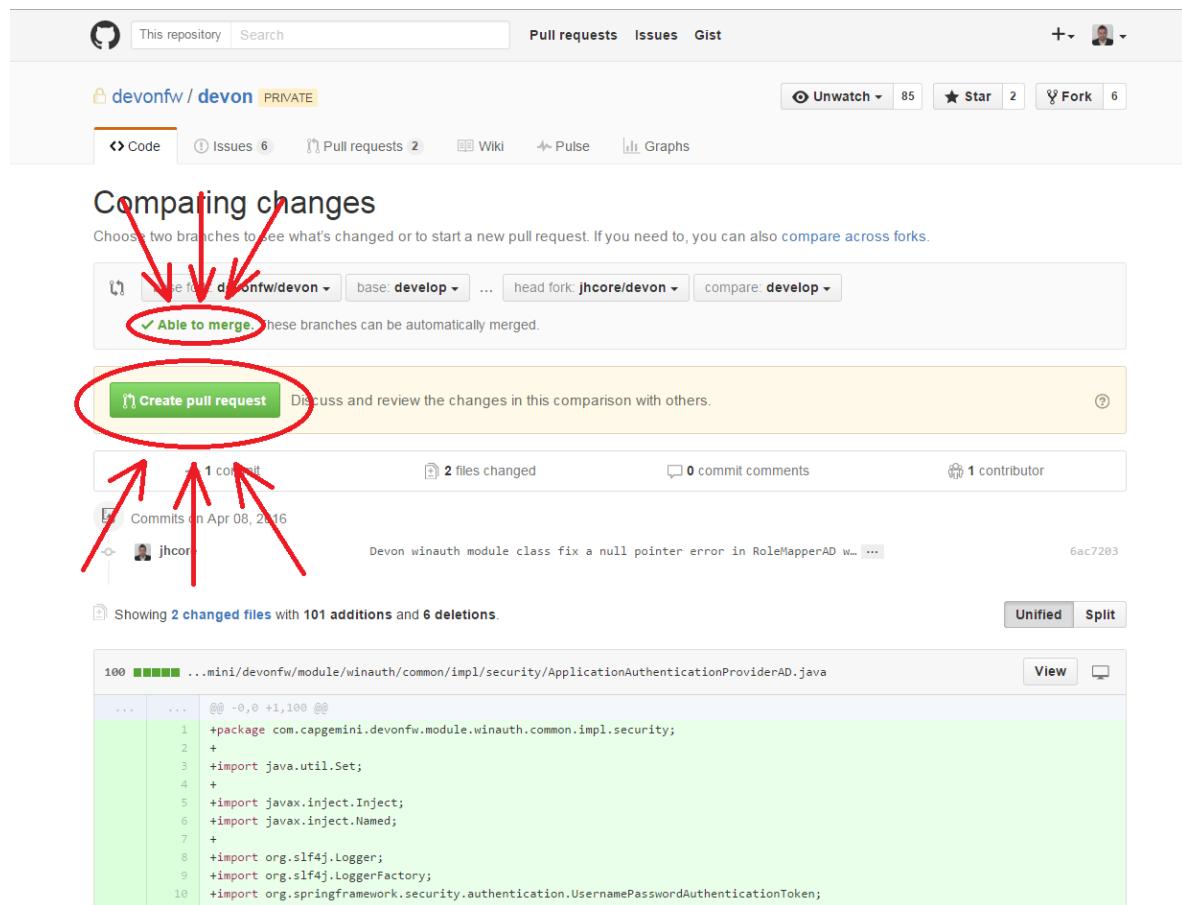
As shown, the `<remote_repository>` can be the URL of the GitHub repository or the name that you defined in the step 3.

[[devon-guide-working-with-git-and-github\_step-7 pull request]] === Step 7 Pull request

At this point, you have the modifications in your remote repository, so you can make a pull request to the remote [Devon / OASP4J](#) repository. To do so, you need to go to your fork repository of [Devon / OASP4J](#), open the branch you want to pull and then press the button "New pull request".



First of all, GitHub will check if the branch is correct and is available to do the pull request. If everything is correct, then you will see as shown in the image below:

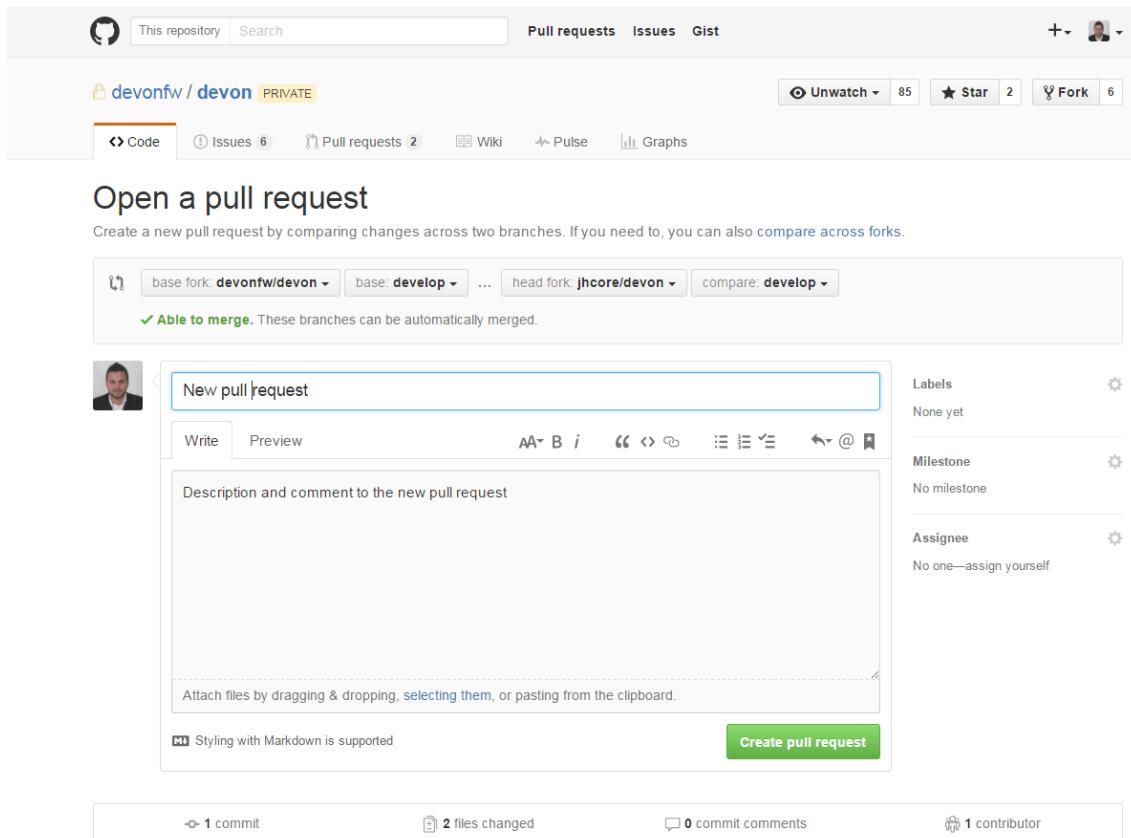


The screenshot shows a GitHub repository comparison between 'devonfw/devon' and 'jhcore/devon'. The 'devonfw/devon' branch is the base, and 'jhcore/devon' is the head fork. The comparison shows 1 commit, 2 files changed, 0 commit comments, and 1 contributor (jhcore). The commit message is: 'Devon winauth module class fix a null pointer error in RoleMapperAD w...'. The code diff shows a single line of Java code being added to ApplicationAuthenticationProviderAD.java:

```
100 100 ...mini/devonfw/module/winauth/common/impl/security/ApplicationAuthenticationProviderAD.java
1 +package com.capgemini.devonfw.module.winauth.common.impl.security;
2 +
3 +import java.util.Set;
4 +
5 +import javax.inject.Inject;
6 +import javax.inject.Named;
7 +
8 +import org.slf4j.Logger;
9 +import org.slf4j.LoggerFactory;
10 +import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
```

As you can see, the branch is available to do the new pull request. Additionally, you can scroll down and look the differences with respect to the original repository.

Check if everything is correct, then you can click "Create pull request" button. Then, you can see a small form with a name of the New pull request and a little description that you need to complete.



When you complete the form, you press the button "Create pull request" and then the pull is sent to be checked and added in the remote original repository.

[[devon-guide-working-with-git-and-github\_step-8 synchronize the repository]] == Step 8 Synchronize the repository

When your Pull request is included in the original repository, you need to actualize your local and remote repository with the original repository. To do so, first of all, you need to check that you are in the development branch.

```
git checkout develop
```

Now, you need to pull the original [Devon / OASP4J](#) repository to your local repository. To do this, you can execute the following command:

```
git pull devon/oasp develop:develop
```

As you can see, you can use the defined variables with the url of [Devon / OASP4J](#) (Step 3) or just the URL of the repository.

When you have the local repository synchronized, you need to push the local development branch to your remote development branch

```
git push <your_git_user> develop:develop
```

As it is commented above, <your\_git\_user> is the variable defined with the URL of your remote repository (the fork of [Devon / OASP4J](#)) (Step 3).

[[devon-guide-working-with-git-and-github\_oasp-issue work]] == OASP Issue Work

[[devon-guide-working-with-git-and-github\_issue-creation and resolution]] === Issue creation and resolution

### 5.2.1 Issue creation

You can create an issue [here](<https://github.com/oasp/oasp4j/issues/new>). Please consider the following points:

- If your issue is related to a specific building block (like e.g. oasp4js), open an issue on that specific issue tracker. If you're unsure which building block is causing your problem, open an issue on this repository.
- Put a label on the issue to mark, whether you suggest an enhancement, report an error or something else.

When reporting the errors:

- Include the version of OASP4j you are using.
- Include screenshots, stack traces.
- Include the behavior you expected.
- Using a debugger, you might be able to find the cause of the problem and you could be the one to contribute a bug-fix.

[[devon-guide-working-with-git-and-github\_preparation-for issue resolution]] === Preparation for issue resolution Before you start working on an issue, check out the following points:

- Try to complete all the other issues, you are working on before. Only postpone issues where you are stuck and consider giving them back in the queue (backlog).
- Check that, no-one else is already assigned or working on the issue.
- Read through the issue and check that you understand the task completely. Collect any remaining questions and clarify them with the one responsible for the topic.
- Ensure that, you are aware of the branch on which the issue shall be fixed and start your work in the corresponding workspace.
- If you are using git, perform your changes on a feature branch.

[[devon-guide-working-with-git-and-github\_definition-of done]] === Definition of Done

- The actual issue is implemented (bug fixed, new feature implemented, etc.).
- The new situation is covered by tests (according to test strategy of the project e.g. for bugs, create a unit test first proving the bug and running red, then fix the bug and check that the test gets green, for new essential features create new tests, for GUI features do manual testing).
- Check the code-style with sonar-Qube in Eclipse. If there are any anomalies in the new or modified code, please rework.
- Check out the latest code from the branch you are working on (svn update, git pull after git commit).
- Make sure that, all the builds and tests are working correctly (mvn clean install).

- Commit your code (svn commit, git push) - for all your commits, ensure that you should stick to the conventions for code contributions (see [code contribution](#)) and provide proper comments (see [coding conventions](#)).
- If no milestone was assigned, please assign suitable milestone.
- Set the issue as done.

### 5.2.2 Code Contribution

We are looking forward to your contribution to OASP4J. This page describes the few conventions to follow. Please note that this is an open and international project and all content has to be in (American) English language.

For contributions to the code, please consider:

- All works on the issue-based follow-up, so check if there is already an issue in the tracker for the task you want to work on or create a new issue for it.
- In case of more complex issues, please get involved with the community and ensure that there is a common understanding of what and how to do it. It is better, not to invest into something that will later be rejected by the community.
- Before you get started, ensure that you comment the issue accordingly and you are the person assigned to the issue. If there is already someone else assigned, get in contact with him, if you still want to contribute to the same issue. It is better, not to invest into something that is already done by someone else.
- Create a [fork](#) of the repository on github to your private github space.
- Checkout this fork and do modifications.
- Ensure that, you stick to the .
- Check in features or fixes as individual commits associated with an xref:../issues[issue] using the commit message format:

```
#<issueId>: <describe your change>
```

Then, GitHub will automatically link the commit to the issue. In case, you worked on an issue from a different repository (e.g. change in oasp4j-sample due to issue in oasp4j), use this commit message format:

```
oasp/<repository>#<issueId>: <describe your change>
```

As an example:

```
oasp/oasp4j#1: added REST service for tablemanagement
```

- If you completed your feature (bugfix, improvement, etc.), use a [pull request](#) to give it back to the community.
- Also, see the [documentation](#) guidelines.

## 5.3 Devcon Command Reference



In the introduction to Devcon we mentioned that Devcon is a tool based on modules that group commands so the different functionalities are stored in these modules that act as utilities containers. The current version of devcon has been released with the following modules

- dist
- doc
- github
- help
- oasp4j
- oasp4js
- project
- sencha
- system
- workspace

### Note

in your Devcon version more modules may have been included. You can list them using the option  
`devon -h`

### 5.3.1 dist

The *dist* module is responsible for the tasks related with the distribution which means all the functionalities surrounding the configuration of the Devon distribution, including the obtention of the distribution itself.

#### **dist install**

The *install* command downloads a distribution from a Team Forge repository and after that extracts the file in a location defined by the user.

#### **dist install requirements**

A user with permissions to download files from Team Forge repository.

#### **dist install parameters**

The *install* parameter needs four parameters to work properly:

- **user**: a Team Forge user with permissions to download files from the repository at least.
- **password**: the Team Forge user password.
- **path**: the path where the distribution must be downloaded.
- **type**: the type of distribution. The options are 'oaspide' to download a oasp4j based distribution or 'devondist' to download a Devon based distribution.

#### ***dist install* example of usage**

A simple example of usage for this command would be the following

```
---
D:\>devon dist install -user john -password 1234 -path D:\Temp\MyDistribution -type devondist
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
[INFO] installing distribution...
[INFO] Downloading Devon-dist_2.0.0.7z (876,16MB). It may take a few minutes.
[=====] 100% downloaded
[INFO] File downloaded successfully.
[...]
[INFO] extracting file...
[INFO] File successfully extracted.
[INFO] The command INSTALL has finished successfully
---
```

You must have in mind that this process can take a while, specially depending on your connection to the internet.

After downloading and installing the distribution successfully installed. You can now follow the manual steps as described in the Devonfw Guide or, alternatively, run 'devon dist init' to initialize the distribution.

#### ***dist init***

The *init* command initializes a newly downloaded distribution.

#### ***dist init* requirements**

A new, not initialized distribution (running it on a configured distribution has no adverse side-effects).

#### ***dist init* parameters**

The *init* parameter needs one parameter to work properly:

- **path**: location of the Devon distribution (current dir if not given).

#### ***dist s2***

The s2 command has been developed to automate the configuration process to use Devon as a Shared Service. This configuration is based on launching two scripts included in the Devon distributions, the *s2-init.bat* and the *s2-create.bat*. The *\*s2-init.bat\** is responsible for configuring the *settings.xml* file (located in the *conf/.m2* directory). Basically enables the connection of *Maven* with the *Artifactory* repository, where the Devon IP modules are stored, and adds the user credentials for this connection.

The *\*s2-create.bat\** creates a new project in the workspace of the distribution, and optionally does a checkout of a Subversion repository inside this new project. Finally the script creates a Eclipse *.bat* starter related to the new project.

### dist s2 requirements

- The command can be launched from any directory within a Devon distribution version 2.0.1 or higher. The Devon distribution is defined by having a *settings.json* file located in the *conf* directory. This file is a JSON object that defines parameters like the version of the distribution or the type which should be *devon-dist* as is showed below.

```
{"version": "2.0.1", "type": "devon-dist"}
```

- An *Artifactory* user with permissions to download files from the repository.
- In case the optional checkout A Subversion user with permissions to do the checkout of the project specified in the *url* parameter.

The command will search for this file to get the root directory where the scripts are located so is necessary to have this file in its correct location.

Apart from this the *settings.xml* file needs to be compatible with the Shared Services autoconfiguration script (*s2-init.bat*).

### dist s2 parameters

So the s2 command needs six parameters to be able to complete the two phases:

- **user**: the userId for Artifactory provided by S2 for the project.
- **pass**: the password for Artifactory.
- **engagementname**: the name of the repository for the engagement.
- **cias**: if the *settings.xml* must be configured for ClaaS user must set this as TRUE. Is an optional parameter with FALSE as default value.
- **projectname**: the name for the new project.
- **svnuser**: the user for the SVN.
- **svnpass**: the password for the SVN.
- **svnurl**: the url for the SVN provided by S2.

### dist s2 example of usage

A simple example of usage for this command would be the followings:

If we only want to configure the *settings.xml* file without using the svn option the simplest usage would be

```
D:\devon-dist\workspaces>devon dist s2 -user john -pass ZMF4AgyhQ5X6Sr9Bd1ohjWcFjL -engagementname
myEngagement -projectname TestProject
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
[...]
INFO: Completed
Eclipse preferences for workspace: "TestProject" have been created/updated
Created eclipse-TestProject.bat
Finished creating/updating workspace: "TestProject"
```

After this the *conf/.m2/settings.xml* file should have been configured and a new (and empty) *TestProject* directory must have been created in the *workspaces* directory and in the distribution root a new *eclipse-testproject.bat* script must have been created too.

We also can get the same result and configure the *settings.xml* for ClaaS using the *ciaas* parameter

```
D:\devon-dist\workspaces>devon dist s2 -user john -pass ZMF4AgyhQ5X6Sr9Bd1ohjWcFjL -engagementname  
myEngagement -projectname TestProject -ciaas true
```

Using the *svn* option to automate the check out from the repository the usage would be

```
D:\devon-dist\workspaces>devon dist s2 -user john -pass ZMF4AgyhQ5X6Sr9Bd1ohjWcFjL -engagementname  
myEngagement -projectname TestProject -svnurl https://coconet...Project/ -svnuser john_svn -svnpass  
12345  
Hello, this is Devcon!  
Copyright (c) 2016 Capgemini  
[...]  
[INFO] The checkout has been done successfully.  
[INFO] Creating and updating workspace...  
[...]  
INFO: Completed  
Eclipse preferences for workspace: "TestProject" have been created/updated  
Created eclipse-TestProject.bat  
Finished creating/updating workspace: "TestProject"
```

After this the *conf/.m2/settings.xml* file should have been configured and a new *TestProject* directory must have been created in the *workspaces* directory with all the files checked out from the *svn* repository and in the distribution root a new *eclipse-testproject.bat* script must have been created too.

### dist info

The *info* command provides very basic information about the Devon distribution, like type, version and path.

#### dist info parameters

The *dist info* command has one optional parameter:

- **path:** path to the distro. Uses current directory if not specified.

### 5.3.2 doc

With this module we can access in a straightforward way to the documentation to get started with Devon framework. The commands of this module show information related with different components of Devon even opening in the default browser the sites related with them.

- **doc devon:** Opens the Devon site in the default web browser.
- **doc devonguide:** Opens the Devon Guide in the default web browser.
- **doc getstarted:** Opens the 'Getting started' guide of Devon framework.
- **doc links:** Shows a brief description of Devon framework and lists a set of links related to it like the public site, introduction videos, the Yammer group and so forth.
- **doc oasp4jguide:** Opens the OASP4J guide.
- **doc sencha:** Opens the Sencha Ext JS 6 documentation site.

### 5.3.3 github

This module is implemented to facilitate getting the Github code from OASP4J and Devon repositories. It has only two commands, one to get the OAPS4J code and other to get the Devon code.

## github oasp4j

This command clones the oasp4j repository to the path that the user specifies in the parameters.

### github oasp4j parameters

The oasp4j command needs only one parameter:

- **path:** the location where the repository should be cloned.
- **proxyHost:** Host parameter for optional Proxy configuration.
- **proxyPort:** Port parameter for optional Proxy configuration.

### github oasp4j example of usage

A simple example of usage for this command would be the following

```
D:\Projects\oasp4j>devon github oasp4j
```

Or using the **-path** parameter

```
D:\>devon github oasp4j -path C:\Projects\oasp4j
```

Also we can define, if necessary, a proxy configuration. The following example shows how to configure the connection for Capgemini's proxy in Europe

```
D:\Projects\oasp4j>devon github oasp4j -proxyHost 1.0.5.10 -proxyPort 8080
```

## github devoncode

This command clones the Devon repository to the path specified in the path parameter.

### github devoncode requirements

A github user with download permissions over the Devon repository.

### github devoncode parameters

The devoncode command needs three parameters:

- **path:** the location where the repository must be cloned.
- **username:** the github user (with permission to download).
- **password:** the password of the github user.
- **proxyHost:** Host parameter for optional Proxy configuration.
- **proxyPort:** Port parameter for optional Proxy configuration.

### github devoncode example of usage

A simple example of usage for this command would be the following

```
D:\>devon github devoncode -path C:\Projects\devon -user John_g -pass 12345
```

Also we can define, if necessary, a proxy configuration. The following example shows how configure the connection for Capgemini's proxy in Europe

```
D:\>devon github devoncode -path C:\Projects\devon -user John_g -pass 12345 -proxyHost 1.0.5.10 -proxyPort 8080
```

### 5.3.4 help

The help module is responsible for showing the help info to facilitate the user the knowledge to use the tool. It has only one command, the *guide* command, that doesn't need any parameter and that basically prints a summary of the devcon general usage with a list of the global options and a list with the available modules

#### **help example of usage**

```
D:\>devon help guide
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: devon <><> [parameters...]
Devcon is a command line tool that provides many automated tasks around
the full life-cycle of Devon applications.
-h,--help      show help info for each module/command
-v,--version    show devcon version
List of available modules:
> help: This module shows help info about devcon
> sencha: Sencha related commands
> dist: Module with general tasks related to the distribution itself
> doc: Module with tasks related with obtaining specific documentation
> github: Module to create a new workspace with all default configuration
> workspace: Module to create a new workspace with all default configuration
```

If you have follow this guide you can realize that the result is the same that is shown with other options as devon or devon -h. This is because these options internally are using this module *help*.

### 5.3.5 oasp4j

This module groups all the devcon functionalities related to the server applications like creating, running and deploying server applications based on the OASP4J project.

#### **oasp4j create**

This command creates a new server project based on the OASP4J archetype.

#### **oasp4j create requirements**

This command needs to be launched from within (or pointing to) a Devonfw distribution.

In a second term internally this command uses the *Maven* plugin included in the Devonfw distributions so in order to be able to use this plugin we should launch this command from a Devonfw command line (use the *console.bat* included in the Devonfw distributions).

#### **oasp4j create parameters**

This command uses five parameters (four of them mandatory).

- **servername**: the name for the new server project.
- **serverpath**: the location for the new server project. Is an optional parameter, if the user does not provide it devcon will use the current directory in its place.

- **packagename**: the name for the project package.
- **groupid**: the groupId for the project.
- **version**: the version for the project.

#### ***oasp4j create* example of usage**

A simple example of usage for this command would be the following

```
D:\devon-dist>devon oasp4j create -servername MyNewProject -packagename
io.devon.application.MyNewProject -groupid io.devon.application -version 1.0-SNAPSHOT
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
[INFO] Scanning for projects...
[...]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.203 s
[INFO] Finished at: 2016-07-14T13:00:17+01:00
[INFO] Final Memory: 10M/42M
[INFO] -----
D:\>
```

Or using the optional **serverpath** parameter to define the location for the project

```
D:\>devon oasp4j create -servername MyNewProject -serverpath D:\devon-dist\ -packagename
io.devon.application.MyNewProject -groupid io.devon.application -version 1.0-SNAPSHOT
```

After that we should have a new *MyNewProject* project created in the *devon-dist* directory.

#### ***oasp4j run***

With this command the user can run a server project application from the embedded tomcat server.

#### ***oasp4j run* requirements**

The command can be launched within a Devon distribution version 2.0.1 or higher. Also verify that your *oasp4j* application has the *devon.json* file well configured.

In case you get a *Detected both log4j-over-slf4j.jar AND bound slf4j-log4j12.jar on the class path, preempting StackOverflowError*.

Seems that there is an error related to the Oasp4j Archetype version 2.1.0 that it's going to be addressed for next releases (more info [here](#)).

However if you find that error you can apply the following workaround:

- Go to the *core/pom.xml* file of your project and comment the dependency:

```
<!--      <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.6.1</version>
      </dependency> -->
```

- Return to your project root directory and execute `mvn install` command.
- Try again with the `devon oasp4j run` command.

## ***oasp4j run* parameters**

The *run* command handles two parameters

- **path**: to indicate the location of the core project of the server app. Is an optional parameter and if not provided by the user devcon will take as the path the directory from which the command has been launched.
  - **port**: the port from which the app should be accessible.

## *oasp4j run* example of usage

A simple example of usage for this command would be the following

Or providing the optional *path* parameter

```
D:\>devon oasp4j run -port 8081 -path D:\devon-dist\workspaces\MyApp\core
```

oasp4i build

With this command the user can build a server project, is the equivalent to the `mvn clean install` command.

## *oasp4j build requirements*

In order to work properly the command must be launched from within (or pointing to) a OASP4J project directory (the `oasp4j` project type is defined in a `devon.json` file with parameter 'type' set to 'oasp4j').

## ***oasp4j build* parameters**

This command only uses one parameter

**-path**: the location of the server project. This is an optional parameter and if the user does not provide it devcon will use in its place the current directory from which the command has been launched.

## *oasp4i build* example of usage

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces\MyApp>devon oasp4j build
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
projectInfo read...
path D:\devon-dist\workspaces\MyApp project type OASP4J

[...]

[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] MyApp ..... SUCCESS [ 0.301 s]
[INFO] MyApp-core ..... SUCCESS [ 12.431 s]
[INFO] MyApp-server ..... SUCCESS [ 3.699 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 16.712 s
[INFO] Finished at: 2016-07-15T11:44:00+01:00
[INFO] Final Memory: 31M/76M
[INFO] -----
D:\devon-dist\workspaces\MyApp>
```

Or using the optional parameter *path*

```
D:\>devon oasp4j build -path D:\devon-dist\workspaces\MyApp
```

## 5.3.6 oasp4js

The oasp4js module is responsible for automating the tasks related to the client projects based on Angular.

### ***oasp4js create***

With this command the user can create a basic oasp4js app.

#### ***oasp4js create requirements***

This command must be used within a Devonfw distribution with version 2.0.0 or higher. You can check your distribution's version looking at the conf/settings.json file.

#### ***oasp4js create parameters***

This command accepts two parameters:

- **clientname**: the name for the application.
- **clientpath**: the location for the new application. Is an optional parameter and if not provided by the user devcon will take as the path the directory from which the command has been launched.

#### ***oasp4js create example of usage***

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces>devon oasp4js create -clientname MyOasp4jsApp
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
Creating project MyOasp4jsApp...
Project created successfully. Please launch 'npm install' to resolve the project dependencies.

D:\devon-dist\workspaces>
```

If everything goes right a new directory *MyOasp4jsApp* must have been created containing the basic structure of an *oasp4js* app.

Finally, as the command informs, in order to complete the app installation the user must launch the command `npm install` from within the new app directory to resolve all the needed dependencies.

```
D:\devon-dist\workspaces\MyOasp4jsApp>npm install
```

The user can also use the next command *oasp4js build* to do that last operation.

### ***oasp4js build***

With this command the user can resolve the dependencies of an *oasp4js* app. The *oasp4js build* command is the equivalent to the `npm install` command.

#### ***oasp4js build* parameters**

- **path:** The location of the *oasp4js* app. Is an optional parameter and if not provided devcon will use the current directory from which the command has been launched instead.

#### ***oasp4js build* example of usage**

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces\MyOasp4jsApp>devon oasp4js build
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
path D:\devon-dist\workspaces\MyOasp4jsApp\.project type OASP4JS

npm WARN deprecated gulp-minify-css@1.2.0: Please use gulp-clean-css
npm WARN deprecated gulp-minify-html@1.0.3: Please use gulp-htmlmin
fetchMetadata -> headers  / |#####
fetchMetadata -> network  | |#####
[...]
| | +-- align-text@0.1.4
| | `-- lazy-cache@1.0.4
| +-- right-align@0.1.3
| `-- wordwrap@0.0.2
+-- decamelize@1.2.0
`-- window-size@0.1.4

npm WARN app@0.0.1 No repository field.
npm WARN app@0.0.1 No license field.
```

Or using the optional parameter *path*

```
D:\devon-dist>devon oasp4js build -path D:\devon-dist\workspaces\MyOasp4jsApp
```

### ***oasp4js run***

In order to launch the *oasp4js* apps devcon provides this *run* command that can be launched even without parameters.

#### ***oasp4js run* parameters**

The only parameter needed is the *clientpath* that points to the client app. This is an optional parameter and if not provided devcon will use by default the directory from within the command is launched.

### ***oasp4js run* example of usage**

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces\MyOasp4jsApp>devon oasp4js run
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
Starting application

[11:30:12] Starting 'sync group1:0'...
[11:30:12] Starting 'sync group2:0'...
[11:30:12] Starting 'styles'...
[11:30:12] Starting 'img:sprite'...
[11:30:12] Starting 'sync group3:0'...
[11:30:12] Starting 'ngTemplatesTasksGeneration'...
[11:30:12] Finished 'ngTemplatesTasksGeneration' after 439 µs

...
[11:30:14] Finished 'serve' after 25 ms
[BS] Access URLs:
-----
  Local: http://localhost:9000/
  External: http://10.68.8.41:9000/
-----
  UI: http://localhost:3001
  UI External: http://10.68.8.41:3001
-----
[BS] Serving files from: .tmp
[BS] Serving files from: app
[BS] Serving files from: bower_components/bootstrap/dist
[BS] Watching files...
```

Or using the optional parameter *clientpath*

```
D:\devon-dist>devon oasp4js run -clientpath D:\devon-dist\workspaces\MyOasp4jsApp
```

In both cases after launching the command a browser should be opened and will show the welcome page of the *oasp4js* app.

### ***oasp4js jumpstart***

This command allows users to get the Oasp4js sample app with all its dependencies included. The command downloads a zip file from Teamforge and extracts all its content in the user's environment.

#### ***oasp4js jumpstart* requirements**

A user with permissions to download files from Team Forge repository.

#### ***oasp4js jumpstart* parameters**

- **path:** the location for the oasp4js sample app file. Is an optional parameter, if not provided the current path will be used.
- **user:** a Team Forge user with permissions to download files from the repository at least.
- **password:** the Team Forge user password.
- **angularVersion:** Optional parameter to choose the Angular version in which the sample app is based. The options are '1' to download Oasp4js sample app based on Angular 1 and '2' to download Oasp4js sample app based on Angular 2. When not provided by the user the default value for this parameter will be '1'.

## **oasp4js jumpstart example of usage**

A simple example of usage for this command would be the following

```
D:\>devon oasp4js jumpstart -user john -password 1234
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
[INFO] downloading file...
[INFO] Downloading oasp4js-1.1.0.zip (52,76MB). It may take a few minutes.
[==          ] 29% downloaded

[...]

file unzip : D:\Temp\.oasp4js-1.1.0\npm-shrinkwrap.json
file unzip : D:\Temp\.oasp4js-1.1.0\package.json
file unzip : D:\Temp\.oasp4js-1.1.0\README.md
Done
File successfully downloaded.
```

Or using optional parameters to define a different location and the Angular version

```
D:\>devon oasp4js jumpstart -user john -password 1234 -path D:\my\custom\location -angularVersion 2
```

### **5.3.7 project**

The *project* module groups the functionalities related to the combined server + client projects.

#### **project create**

With this command the user can automate the creation of a combined server and client project (Sencha or oasp4js).

##### **project create requirements**

If you want to use a Sencha app as client you will need a github user with permissions to download the *devon4sencha* repository.

##### **project create parameters**

Basically this command needs the same parameters as the 'subcommands' that is using behind (oasp4j create, oasp4js create, sencha workspace and sencha create)

- **combinedprojectpath**: the path to locate the server and client projects. Is an optional parameter and if not provided by the user devcon will take as the path the directory from which the command has been launched.
- **servername, packagename, groupid, version**: the parameters related to the Server application. You can get more details in the 'oasp4j create' command reference in this document.
- **clienttype**: the type for the client app, you can provide *oasp4js* for Angular based client or *devon4sencha* for Sencha based client.
- **clientname**: the name for the client app.
- **clientpath**: the path to locate the client app. Current directory if not provided.
- **createsenchaws**: is an optional parameter that indicates if the Sencha workspace needs to be created (by default its value is FALSE).

### project create example of usage

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces\combined>devon project create -servername myServerApp -groupid com.capgemini.devonfw -packagename com.capgemini.devonfw.myServerApp -version 1.0 -clientname myClientApp -clienttype oasp4js
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
serverpath is D:\devon-dist\workspaces\combined\.
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[...]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.862 s
[INFO] Finished at: 2016-08-05T09:23:35+01:00
[INFO] Final Memory: 10M/43M
[INFO] -----
Adding devon.json file...
Project Creation completed successfully
Creating client project...
Creating project myClientApp...
Adding devon.json file...
Editing java/pom.xml...
Project created successfully. Please launch 'npm install' to resolve the project dependencies.
Adding devon.json file to combined project...
Combined project created successfully.
```

With this example we have created a Server + Oasp4js app in the D:\devon-dist\workspaces\combined directory. So within this folder we should find:

- *myServerApp* folder with the oasp4j app.
- *myClientApp* folder with the `oasp4js` app.
- the devon.json file with the following configuration:

```
{"version": "2.0.1",
"type": "COMBINED",
"projects": ["myServerApp", "myClientApp"]
}
```

As you can see the 'projects' property points to the 'subprojects' created. In case we had used the *clientpath* parameter to locate it in a different place that 'project' will reflect it pointing to the client path location:

```
{"version": "2.0.1",
"type": "COMBINED",
"projects": ["myServerApp", "D:\devon-dist\otherDirectory\myClientApp"]
}
```

### Other possible usages

- D:\devon-dist\TEST>devon project create -servername sss -groupid com.cap -packagename com.cap.sss -version 1.0 -clientname ccc -clienttype devon4sencha -clientpath D:\devon-dist\TESTB

Will create a server app (sss) in current directory and a Sencha app in the TESTB directory (that must be a Sencha workspace)

- D:\devon-dist\TEST>devon project create -servername sss -groupid com.cap -packagename com.cap.sss -version 1.0 -clientname ccc -clienttype devon4sencha -clientpath D:\devon-dist\TESTB -createsenchaws true

Will create a server app (sss) in current directory and a Sencha workspace with a Sencha app inside in the TESTB directory.

- D:\devon-dist\TEST>devon project create -servername sss -groupid com.cap -packagename com.cap.sss -version 1.0 -clientname ccc -clienttype devon4sencha

Will create a server app (sss) and a Sencha workspace with a Sencha app inside, all in current directory.

## project build

This command will build both client and server project.

### project build requirements

In order to work properly, the command must be launched from within (or pointing to) a Devon distribution (the oasp4j project type is defined in a *devon.json* file with parameter 'type' set to 'oasp4j' in the server project ). The directory from where build command is fired should contain client and server project at same level, and directory should contain a *devon.json* which should have project type as **COMBINED**, and client project should contain a *devon.json* file with parameter 'type' set to 'oasp4js' or 'devon4sencha'.

### project build parameters

The build command takes three parameters and two of them are mandatory.

- **path** : This is an optional parameter. It points to server project workspace and if value of this parameter not given, it takes default value as current directory.
- **clienttype** : This parameter shows which type of client is integrated with server i.e oasp4js or sencha. Its a mandatory one.
- **clientpath** : It should point to client directory i.e where the client code is located. Again a mandatory one.

### project build example of usage

A simple example of usage for this command would be the following

```
D:\>devon project build -path D:\FIN_IDE\oasp4j-ide-all-2.0.0\samplec -clienttype oasp4js -clientpath D:\FIN_IDE\oasp4j-ide-all-2.0.0\clientdoc
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
projectInfo read...
path D:\FIN_IDE\oasp4j-ide-all-2.0.0\samplecproject type OASP4J
Completed
path D:\FIN_IDE\oasp4j-ide-all-2.0.0\clientdocproject type OASP4JS
Completed
```

## project deploy

This command automates all the process described in the [deployment on tomcat](#) section. It creates a new tomcat server associated to the combined server + client project in the *software* directory of the distribution and launches it to make the project available in a browser.

### project deploy requirements

The command automates the packaging of the combined Server + Client project but the user must configure those apps to work properly so you need to verify that:

- The client app *points* to the server app: in Sencha projects the 'server' property of *app/Config.js* or *app/ConfigDevelopment.js* (depending of the type of build) must point to your server app. In case of *oasp4js* projects we will need to configure the *baseUrl* property of the '*config.json*' file to point to our server.
- The server redirects to the client: in the server project the file ...\\serverApp\\server\\src\\main \\webapp\\index.jsp should redirect to *jsclient* profile *index.jsp*

```
<%
  response.sendRedirect(request.getContextPath() + "/jsclient/");
%>
```

- The combined project must have a *devon.json* file defining the type (that must be 'combined') and the subprojects (server and client):

```
{"version": "2.0.1",
"type": "COMBINED",
"projects": [ "D:\\devon-dist\\workspaces\\SenchaWorkspace\\myClientApp" , "myServerApp" ]
}
```

In the example above that *devon.json* file defines a server app (*myServerApp*) that is located within the combined project directory (so we do not need to provide a path, only the folder name) and a client app (*myClientApp*) located in a Sencha workspace outside the combined project directory (so we need to provide the path).

- Each 'subprojects' (server and client) must have its corresponding *devon.json* file well formed (the 'type' must be *oasp4j* for server and for client apps *oasp4js* or *devon4sencha*).
- The command must be launched from within a valid Devonfw distribution.

### project deploy parameters

- **tomcatpath**: the path to the tomcat folder. Devcon will look for the distribution's Tomcat when this parameter is not provided.
- **clienttype**: type of client either angular or Sencha (obtained from 'projects' property in *devon.json* when not given).
- **clientpath**: path to client project (obtained from 'projects' property in *devon.json* when not given).
- **serverpath**: path to server project (obtained from 'projects' property in *devon.json* when not given).
- **path**: path for the combined project (current directory when not given).

### project deploy example of usage

A simple example of usage for this command would be the following

D:\devon-dist\workspaces\MyCombinedProject>devon project deploy Hello, this is Devcon! Copyright (c) 2016 Capgemini

# After Tomcat finishes the loading process the app should be available in: localhost:8080/myServerApp-server-1.0 #

The process will open a new command window for the Tomcat's launching process and finally will show us the url where the combined app should be accessible.

#### Note

The url is formed with the name of the .war file generated when packaging the app

If we use the optional parameter *path*

D:\devon-dist>devon project deploy -path D:\devon-dist\workspaces\MyCombinedProject

#### project run

This command runs the server & client project(unified build) in debug mode that is separate client and spring boot server.

#### project run requirements

Please verify the *oasp4j run* and *oasp4js run* or *sencha run* requirements.

#### project run parameters

- **clienttype** : This parameter shows which type of client is integrated with server i.e oasp4js or sencha and its a mandatory parameter
- **clienttype** : the type of the client app ('oasp4js' or 'devon4sencha').
- **clientpath** : Location of the oasp4js app.
- **serverport** : Port to start server.
- **serverpath** : Path to Server project Workspace (currentDir if not given).

#### project run example of usage

A simple example of usage for this command ( for client type oasp4js) would be the following

```
D:\>devon project run -clienttype oasp4js -clientpath D:\FIN_IDE\oasp4j-ide-all-2.0.0\workspaces\main\examples\oasp4js -serverport 8080 -serverpath D:\FIN_IDE\oasp4j-ide-all-2.0.0\workspaces\main\code\oasp4j\samples\server
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
path before modification D:\FIN_IDE\oasp4j-ide-all-2.0.0\workspaces\main\code\oasp4j\samples\server
Server project path D:\FIN_IDE\oasp4j-ide-all-2.0.0\workspaces\main\code\oasp4j\samples\server
Application started
Starting application
```

After launching the command, a browser should be opened and will show the welcome page of the oasp4js app.

## 5.3.8 sencha

*Sencha* is a pure JavaScript application framework for building interactive cross platform web applications and is the view layer for web applications developed with Devon Framework. This module encapsulates the *Sencha Cmd* functionality that is a command line tool to automate tasks around *Sencha* apps.

### sencha run

This command compiles in DEBUG mode and then runs the internal Sencha web server. Is the equivalent to the *Sencha Cmd*'s *sencha app watch* and does not need any parameter.

#### sencha run requirements

We should launch the command from a Devon4Sencha project which is defined by a *devon.json* file with parameter 'type' set to 'Devon4Sencha'

```
{ "version": "2.0.0",  
  "type": "Devon4Sencha"}
```

#### sencha run example of usage

A simple example of usage for this command would be the following

```
D:\devon-dist\workspaces\senchaProject>devon sencha run
```

### sencha workspace

With this command we can generate automatically a fully functional Sencha workspace in a directory of our machine.

#### sencha workspace requirements

We will need a Github user with permissions to clone the *devon4sencha* repository.

#### sencha workspace parameters

The *sencha workspace* command needs five parameters and four of them are mandatory.

- **path**: the location where the workspace should be created. This parameter is optional and if the user does not provide it devcon will take the current directory as the location for the Sencha workspace.
- **username**: the github user with permission to download the *devon4sencha* repository.
- **password**: the password of the github user.
- **proxyHost**: Host parameter for optional Proxy configuration.
- **proxyPort**: Port parameter for optional Proxy configuration.

#### sencha workspace example of usage

A simple example of usage for this command would be the following

```
D:\>devon sencha workspace -path D:\MyProject -username john -password 1234  
Hello, this is Devcon!  
Copyright (c) 2016 Capgemini  
Cloning into 'D:\MyProject\MySenchaWorkspace'...  
Having repository: D:\MyProject\MySenchaWorkspace\.git
```

So after that we will have a sencha workspace located in the *D:\MyProject* directory.

Also we can define, if necessary, a proxy configuration. The following example shows how to configure the connection for Capgemini's proxy in Europe

```
D:\>devon sencha workspace -path D:\MyProject -username john -password 1234 -proxyHost 1.0.5.10 -proxyPort 8080
```

### **sencha copyworkspace**

With this command we can make create new Sencha workspace by making a copy from an existing Devon dist to a particular path

#### **sencha copyworkspace requirements**

There should be a Devonfw distribution present which included the 'workspaces\examples\devon4sencha' folder

#### **sencha copyworkspace parameters**

The *sencha copyworkspace* command needs two parameters. Both are optional.

- **workspace**: the path to the workspace. This parameter is optional. Devcon will take the current directory if not provide and in that case it will use the name 'devon4sencha'.
- **distpath**: the path to a Devonfw Dist (Current directory if not provided)

### **sencha build**

This command builds a Sencha Ext JS6 project. Is the equivalent to the *Sencha Cmd*'s *sencha app build*.

#### **sencha build parameters**

This command only has one parameter and it is optional

- **appDir**: the path to the app to be built. If the user does not provide it devcon will use the current directory as the location of the Sencha app.

#### **sencha build example of usage**

A simple example of usage for this command would be the following

```
D:\MySenchaWorkspace\MyApp>devon sencha build
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
OUTPUT:Sencha Cmd v6.1.2.15
OUTPUT:[INFO] Processing Build Descriptor : classic
[...]
[INFO] [LOG] Sencha App Watch Started
[INFO] [LOG]Sencha Build Successful
D:\MySenchaWorkspace\MyApp>
```

And using the optional parameter *appDir* to locate the app the usage would be like the following

```
D:\>devon sencha build -appDir D:\MySenchaWorkspace\MyApp
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
OUTPUT:Sencha Cmd v6.1.2.15
OUTPUT:[INFO] Processing Build Descriptor : classic
[...]
[INFO] [LOG] Sencha App Watch Started
[INFO] [LOG]Sencha Build Successful
D:\>
```

## sencha create

This command creates a new Sencha Ext JS6 app.

### sencha create requirements

The command must be launched within a Sencha workspace or pointing to a Sencha workspace using the optional parameter *workspacepath*. So in order to work properly first we will need to have a Sencha workspace ready in our local machine.

### sencha create parameters

The create parameters handles two parameters

- **appname**: the name for the new app.
- **workspacepath**: optionally the user can specify the location of the Sencha workspace. If the user does not provide it the current directory will be used as default.

### sencha create example of usage

A simple example of usage for this command would be the following

```
D:\MySenchaWorkspace>devon sencha create -appname MyNewApp
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
OUTPUT:Sencha Cmd v6.1.2.15
OUTPUT:[INF] Loading framework from D:\MySenchaWorkspace\
[...]
[INFO] [LOG]Sencha Ext JS6 app Created
D:\MySenchaWorkspace>
```

And using the optional parameter *workspacepath* to locate the Sencha workspace the command would be like the following

```
D:\>devon sencha create -appname MyNewApp -workspacepath D:\MySenchaWorkspace
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
OUTPUT:Sencha Cmd v6.1.2.15
OUTPUT:[INF] Loading framework from D:\MySenchaWorkspace\
[...]
[INFO] [LOG]Sencha Ext JS6 app Created
D:\>
```

After that we will have a new Sencha app called *MyNewApp* in our Sencha workspace.

## 5.3.9 workspace

This module handles all tasks related to distribution workspaces.

### workspace create

This command automates the creation of new workspaces within the distribution with the default configuration including a new Eclipse .bat starter related to the new project.

### workspace create parameters

The create command needs two parameters:

- **devonpath**: the path where the devon distribution is located.

- **foldername**: the name for the new workspace.

#### **workspace create example of usage**

A simple example of usage for this command would be the following

```
D:\>devon workspace create -devonpath C:\MyFolder\devon-dist -foldername newproject
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
[INFO] creating workspace at path D:\devon2-alpha\workspaces\newproject
[...]
```

As a result of that a new folder *newproject* with the default project configuration should be created in the *C:\MyFolder\devon-dist\workspaces* directory alongside an *eclipse-newproject.bat* starter script in the root of the distribution.

### **5.3.10 system**

This module contains system wide commands related to devcon.

#### **system install**

This command installs devcon on user's HOME directory or at an alternative path provided by user.

It should be used as a very first step to install Devcon, [see more here](#)

```
> java -jar devcon.jar system install
```

If you are behind a proxy you must configure the connection using the optional parameters **-proxyHost** and **-proxyPort**. In following example we show how to use the *system install* command for Capgemini's proxy in Europe

```
> java -jar devcon.jar system install -proxyHost 1.0.5.10 -proxyPort 8080
```

#### **system update**

Launching this command the user can update the Devcon version installed to the last version available.

#### **system update example of usage**

A simple example of usage for this command would be the following

```
D:\>devon system update
```

As occurs with the *system install* command, if you are behind a proxy you will need to use the optional parameters **-proxyHost** and **-proxyPort** to configure the connection. The following example shows how to configure the *system update* with the Capgemini's proxy in Europe

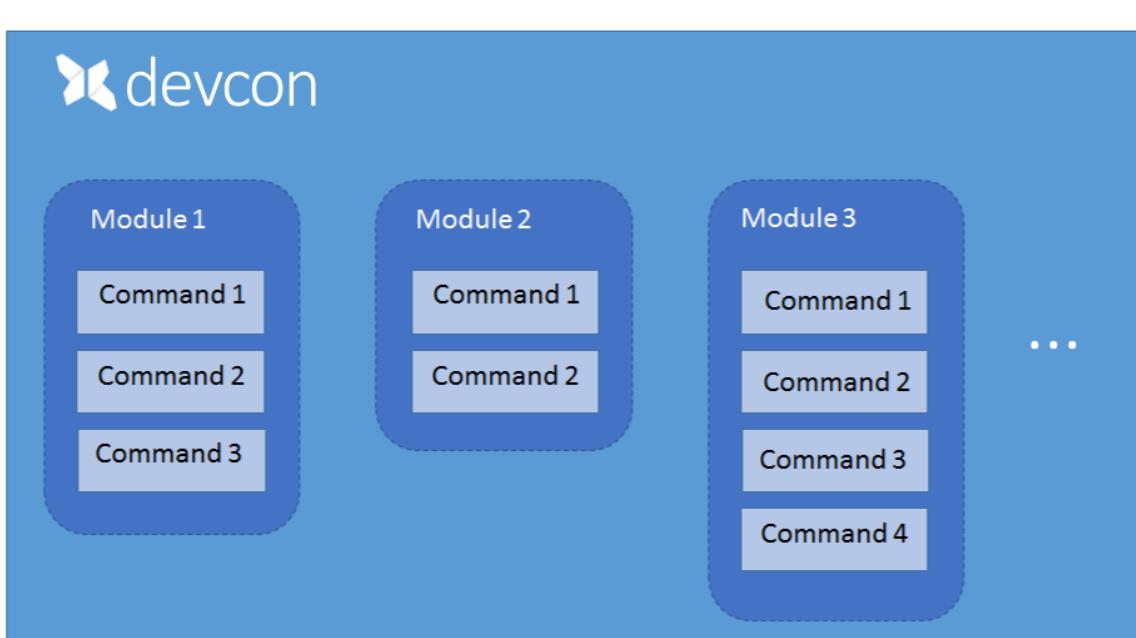
```
D:\>devon system update -proxyHost 1.0.5.10 -proxyPort 8080
```

## 5.4 Devcon Command Developer's guide

### 5.4.1 Introduction

[Devcon](#) is a cross-platform command line and GUI tool written in Java that provides many automated tasks around the full life-cycle of Devonfw applications.

The structure of Devcon is formed by two main elements: *modules* and *commands*.



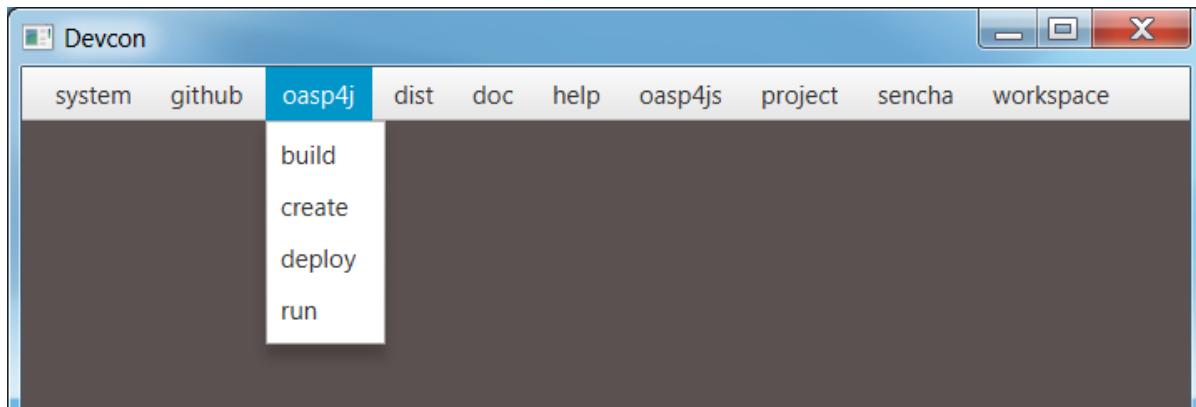
where each module represents an area of Devon and groups commands that are related to some specific task of that area.

There is also a third element with a main spot in Devcon, the *parameters*, we will see them later.

After [installing Devcon](#) you can see the modules and commands available out of the box opening a command line and using the command `devon -g` to launch the Devcon's graphic user interface.

#### Note

Using the command line and the command `devon -h` (even using only the keyword `devon` or `devcon`) and `devon <module> -h` will show the equivalent information.



The available modules appear in the window bar and clicking over each module a drop down menu shows the list of commands grouped under a particular module.

As showed above the module *oasp4j* has four commands related to the *oasp4j* projects: *create*, *build*, *run* and *deploy*. Each command takes care of an specific task within the context of that particular module.

[[devcon-command-developers-guide\_creating-your own devcon modules]] == Creating your own Devcon modules

Devcon has been designed to be easily extended with new custom functionalities. Thanks to its structure based on *modules* and *commands* (and *parameters*) the users can cover new tasks simply including new modules and commands to the tool.

We will be able to do that in two ways:

- Adding a new Java module in the *core* of Devcon.
- Adding an external module written in Javascript.

Let's see the basic elements to have in mind before starting with the addition of new modules.

[[devcon-command-developers-guide\_elements-and their keywords]] === Elements and their keywords

Each main element of Devcon needs to be *registered* to become accessible, to achieve that we annotate each element with a specific *keyword* that will tell Devcon, during the launching process, which elements are available as modules and commands.

### **module registry**

Internally the modules are registered in Devcon's context using the `@CmdModuleRegistry` annotation and providing some *metadata* (like *name*, *description*, etc.) to define the basic details of the module.

In de Javascript approach this annotation will be replaced by a *json* file.

### **command registry**

In the same way, the commands in Devcon are registered using the `@Command` annotation, that also allows to add *metadata* (*name*, *description*, etc.) to provide more information.

### **parameter registry**

In most cases the commands will need parameters to work with. The `@Parameters` and `@Parameter` annotations allow to register those in Devcon. The `@Parameter` annotation also allows to define the basic info of each parameter (*name*, *description*, etc.).

## [[devcon-command-developers-guide\_creating-a java module]] === Creating a java module

If you are not interested in create *core* modules and want to focus on external Javascript modules skip this section and go directly to [Javascript modules](#) part.

So once we have the basic definition of the Devcon's elements and we know how to register them, let's see how to add a new module in Devcon's *core* using Java.

In this example we are going to create a new module called *file* in order to manage files. As a second stage we are going to add an *extract* command to extract zip files. To avoid the tricky details we are going to reuse the *unzip* functionality already implemented in the Devcon's utilities.

1 - Get the last Devcon release from <https://github.com/devonfw/devcon/releases>

2 - Unzip it and *Import* the Devcon project using Eclipse.

3 - In `src/main/java/com.devonfw.devcon/modules` folder create a new package *file* for the new module and inside it add a new *File* class.

### Module annotations

To define the class as a Devcon module we must provide:

- **@CmdModuleRegistry** annotation with the attributes:
  - *name*: for the module name.
  - *description*: for the module description that will be shown to the users.
  - *visible*: if not provided its default value is *true*. Allows to hide modules during develop time.
  - *sort*: to sort modules, if not provided the default value will be *-1*. If *sort >= 0*, it will be sorted by descending value. Modules which do not have any value for *sort* attribute or which have value *<1* will be omitted from numeric sort and will be sorted alphabetically. This modules will be appended to the modules which are sorted numerically.
- extend the *AbstractCommandModule* to have access to all internal features already implemented for the modules (access to output and input methods, get metadata from the project *devon.json* file, get the directory from which the command has been launched, get the root of the distribution and so forth).

Finally we will have something like

```
@CmdModuleRegistry(name = "file", description = "custom devcon module", sort = -1)
public class File extends AbstractCommandModule {

    ...
}
```

### Command annotations

Now is time to define the command *extract* of our new module *file*. In this case we will need to provide:

- **@Command** annotation with attributes:
  - *name*: for the command name.

- *description*: for the command description that will be shown to the users.
- *context*: the context in which the command is expected to be launched regarding a project. E.g. think in the *oasp4j run* command. In this case the *run* command of the *oasp4j* module needs to be launched within the context of an *oasp4j* project. We will define that context using this *context* attribute. The options are:
  - *NONE*: if the command doesn't need to be launched within a project context.
  - *PROJECT*: if the command is expected to be launched within a project (*oasp4j*, *oasp4js* or *Sencha*). In these cases this context definition will automatically provide a default *path* parameter to the command parameters alongside some extra project info (see the *oasp4j run* implementation.).
  - *COMBINEDPROJECT*: if the command needs to be launched within a combined (server & client) project.
  - *proxyParams*: in case you need to configure a proxy this attribute will inject automatically a *host* and *port* parameters as part of the parameters of your command.
  - *sort*: see the *sort* attribute in the previous section.

## Parameter annotations

To define the parameters of our *extract* method we must use the following annotations:

- **@Parameters** annotation to group the command parameters
  - *value*: an array with the parameters
    - **@Parameter** annotation for each parameter expected.
      - *name*: the name for the parameter.
      - *description*: the description of the parameter to be shown to the users.
      - *optional*: if the parameter is mandatory or not, by default this attribute has as value *false*, so by default a parameter will be mandatory.
      - *sort*: see the *sort* attribute in the previous section.
    - *inputType*: the type of field related to the parameter to be shown in the graphic user interface of Devcon.
      - *GENERIC* for text field parameters.
      - *PATH* if you want to bind the parameter value to a *directory window*.
      - *PASSWORD* to show a password field.
      - *LIST* to show a dropdown list with predefined options as value for a parameter.

Let's imagine that in our *extract* example we are going to define two parameters *filepath* and *targetpath* (the location of the zip file and the path to the folder to store the extracted files). As our command will extract a zip file we don't need a particular project context so we will use the *ContextType.NONE*.

Finally, importing the package `com.devonfw.devcon.common.utils.Extractor` we will have access to the `unZip` functionality. Also, thanks to the `AbstractCommandModule` class that we have extended we have access to an output object to show info/error messages to the users.

So our example will look like

```
@CmdModuleRegistry(name = "file", description = "custom devcon module", sort = -1)
public class File extends AbstractCommandModule {

    @Command(name = "extract", description = "This command extracts a zip file.", context =
    ContextType.NONE)
    @Parameters(values = {
        @Parameter(name = "filepath", description = "path to the file to be extracted", inputType =
        @InputType(name = InputTypeNames.GENERIC)),
        @Parameter(name = "targetpath", description = "path to the folder to locate the extracted files",
        inputType = @InputType(name = InputTypeNames.PATH)) })
    public void extract(String filepath, String targetpath){
        getOutput().showMessage("Extracting...");
        try {
            Extractor.unZip(filepath, targetpath);
            getOutput().showMessage("Done!");
        } catch (Exception e) {
            getOutput().showError("Ups something went wrong.");
        }
    }
}
```

[[devcon-command-developers-guide\_generate-the-jar]] ===== Generate the jar

Finally, we need to generate a new `devcon.jar` file containing our new module. To do so, in Eclipse, with right click over the `devcon` project in the *Project Explorer* panel:

- *Export > Runnable JAR file > Next*
- Runnable JAR File Export window:
  - Launch configuration: Devcon (if you don't have any option for that parameter try to launch once the `Devcon.java` class with right click and *Run as > Java Application* and start again the JAR generation).
  - Export destination: select a location for the jar.
  - Check 'Extract required libraries into generated JAR'.
  - Click *Finish* and click *OK* in the next window prompts.

Once we have the `devcon.jar` file we have two options depending if we are customizing a Devcon installed locally or the Devcon tool included with the Devon distributions (from version 2.1.1 onwards).

- OPTION1: If you are working over a local installation of Devcon you only need to copy the `devcon.jar` you just created, to `C:\Users\{Your User}\.devcon` replacing the `devcon.jar` that is inside of that directory with your new `devcon.jar` (be aware that the directory `.devcon` may be placed in another drive like `D`).

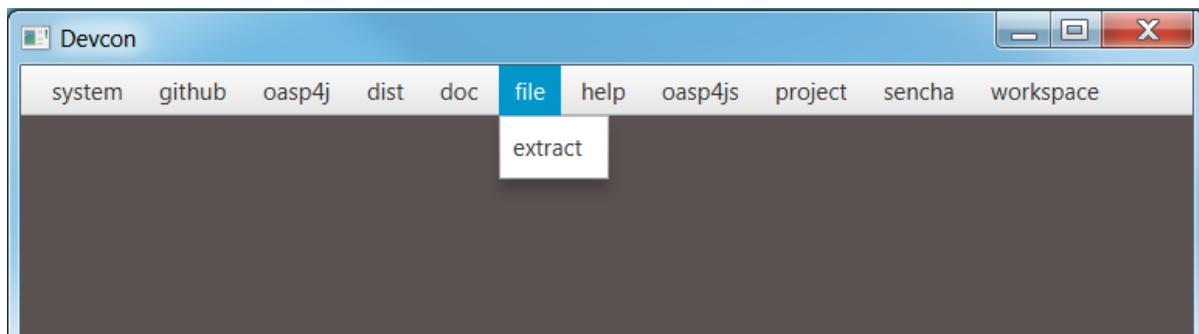
### Note

If you don't have Devcon installed you can see how to install it [here](#)

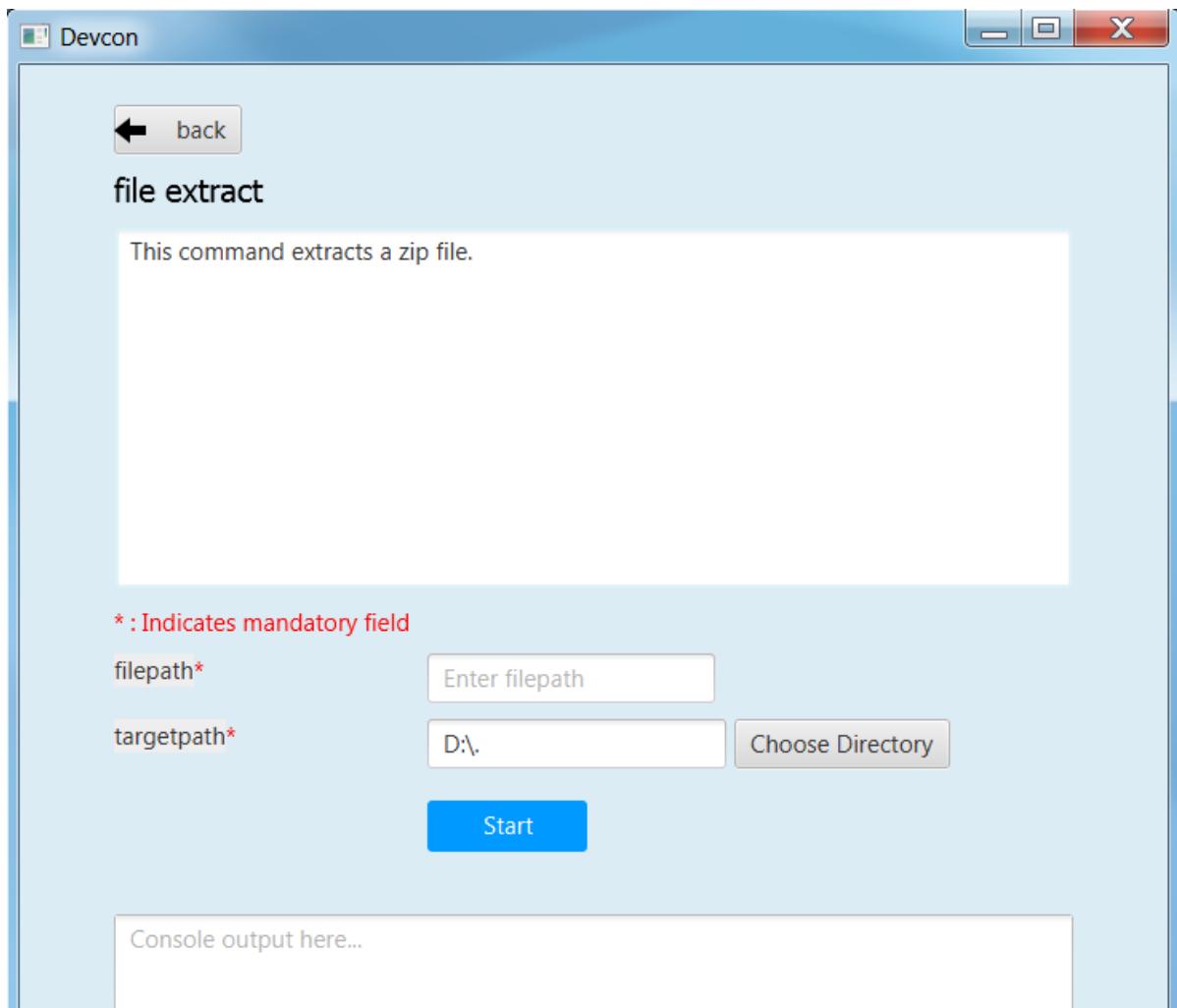
- OPTION 2: In case you are working over the copy of Devcon enabled by default in Devon distributions you only need to copy the *devcon.jar* you just created, to *Devon-distribution\software\devcon* replacing the *devcon.jar* that is inside of that directory with your new *devcon.jar*

Once we have installed our customized version of Devcon we can open the Windows command line (for local Devcon installations) or *console.bat* script (for the Devcon included in Devon distributions) and type *devcon -g* or *devcon -h*. The first one will open the Devcon graphic user interface, the second one will show the Devcon basic info in the command line. In both cases we should see our new module as one of the available modules.

In case of the *gui* option we will see



And selecting the *extract* command we can see that the parameters we defined appear as mandatory parameters.



### Note

If you want to try the same but using the command line you can use the command `devcon file extract -h`

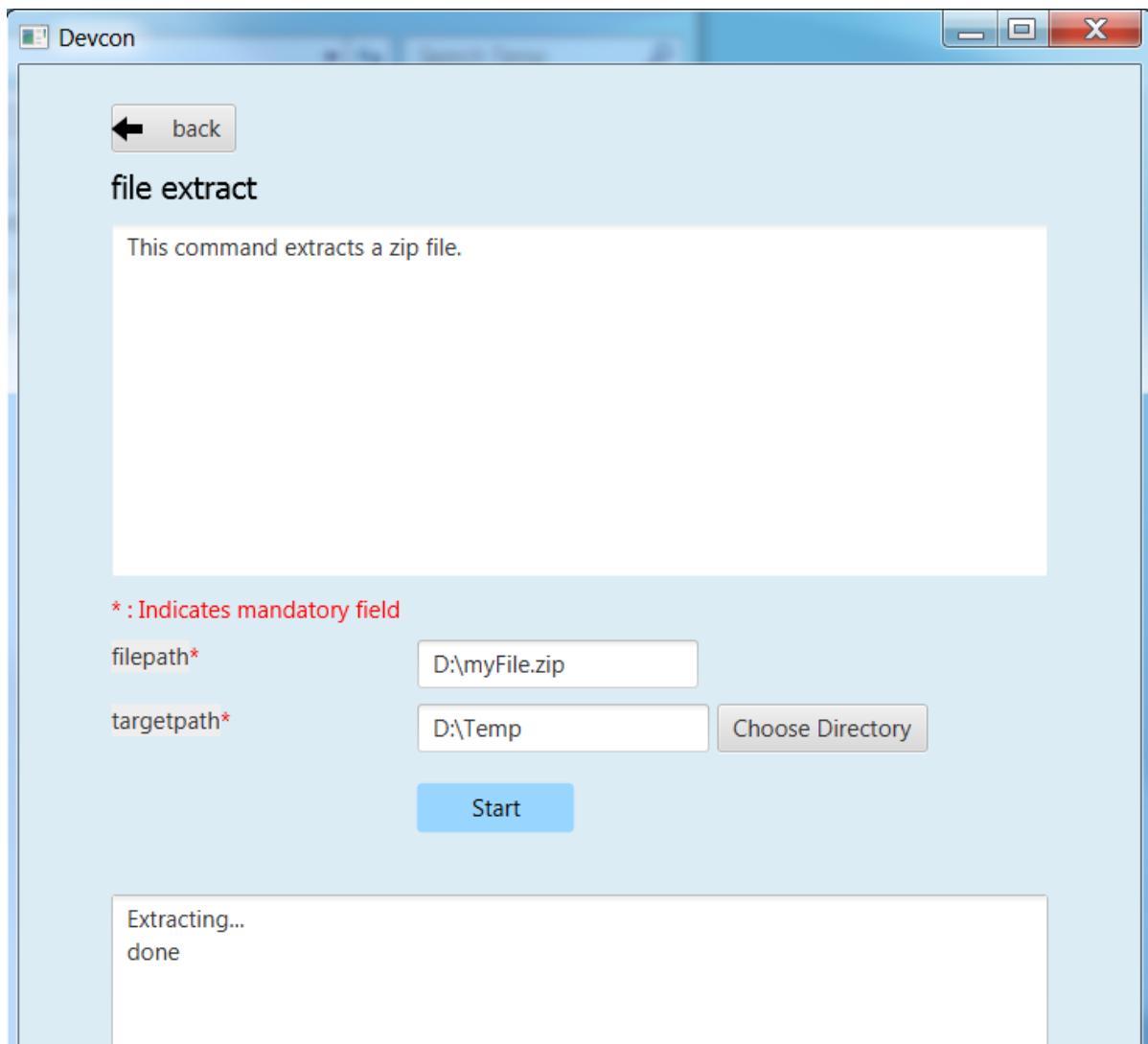
[[devcon-command-developers-guide\_using-our module and command]] ===== Using our module and command

Finally we want to use the `extract` command of our `file` module to extract a real zip file.

We have a `myFile.zip` in `D:` and want to extract the files into `D:\Temp` directory

[[devcon-command-developers-guide\_with-the gui]] ===== with the gui

We will need to provide both mandatory parameters and click `Start` button



[[devcon-command-developers-guide\_with-the command line]] ===== with the command line

We would obtain the same result using the command line

```
C:\>devcon file extract -filepath D:\myFile.zip -targetpath D:\Temp
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
Extracting...
file unzip : D:\Temp\myFile\file1.txt
file unzip : D:\Temp\myFile\file2.txt
file unzip : D:\Temp\myFile\file3.txt
file unzip : D:\Temp\myFile\file4.txt
Done

C:\>
```

That's all, with these few steps, we have created and included a new customized module written in Java in the Devcon's core.

### Javascript modules

As we mentioned at the beginning of this chapter Devcon allows to be extended with custom modules in an external way by adding modules written in Javascript.

### Note

You will need to have installed Java 8 to be able to run Javascript modules.

We have seen how to define the Devcon's elements (modules, commands and parameters) and how to register them (using keywords) so let's see how to add a new module to Devcon using Javascript.

### Module structure

The Javascript modules must include two main files:

- the **commands.json** file that contains the definition of the elements of the module (module metadata, commands and parameters).
- a Javascript file **<name of the module>.js** with the logic of the module.

[[devcon-command-developers-guide\_how-to register a module]] ===== How to register a module

To register a Javascript module we only need to create a directory with that two files and add it to the Devcon's module engine. If you have installed Devcon locally you should add that directory in a *scripts* directory within the `C:\Users\{Your User}\.devcon` folder but if you are customizing the Devcon included by default in the Devon distributions (for versions 2.1.1 or higher) you should add the directory with the *json* and the *js* files in a *scripts* directory within the `Devon-dist\software\devon` folder (we will see it later in more detail).

### Module definition

The *commands.json* file located in the Javascript module folder defines the elements included in it, from the module details, as name or description, to the commands and its parameters.

If you have followed the [Creating a Java module](#) section you have seen that for the Java modules we use the `@CmdModuleRegistry` annotation to register a module. In the case of the Javascript modules this is replaced by the *commands.json* file itself so we won't have an equivalent *module registry* keyword.

To define the module in the *commands.json* file we can use the following attributes:

- *name*: for the module name.
- *description*: for the module description that will be shown to the users.
- *visible*: `true/false` attribute. Allows to hide modules in case we don't want them to be available.
- *sort*: to sort modules, if *sort*  $\geq 0$ , it will be sorted by descending value. Modules which have value  $< 1$  will be omitted from numeric sort and will be sorted alphabetically. This modules will be appended to the modules which are sorted numerically.

An example for a *commands.json* might look like

```
{  
  "name": "myJSmodule",  
  "description": "this is an example of a Devcon Javascript module",  
  "visible": true,  
  "sort": -1,  
  ...  
}
```

## Command definition

Also in the `commands.json` file we will define the commands of the module and its parameters.

- We will use a **commands** array to enumerate all the commands of a module. Each command will be defined with the following attributes:
  - *name*: for the command name.
  - *path*: path to the *js* file that contains the logic of the module. If this is located in the same folder than the `commands.json` file we can provide only the name of the file, without the path.
  - *description*: for the command description that will be shown to the users.
  - *context*: the context in which the command is expected to be launched regarding a project. E.g. the *run* command of the *oasp4j* module needs to be launched within the context of an *oasp4j* project. The options to define the context are:
    - *NONE*: if the command doesn't need to be launched within a project context.
    - *PROJECT*: if the command is expected to be launched within a project (*oasp4j*, *oasp4js* or *Sencha*). In these cases this context definition will automatically provide a default *path* parameter to the command parameters alongside some extra project info (see the *oasp4j run* implementation.).
    - *COMBINEDPROJECT*: if the command needs to be launched within a combined (server & client) project.
  - *proxyParams*: in case your command needs to configure a proxy, this attribute will inject automatically a *host* and *port* parameters as part of the parameters of your command.
  - *sort*: see the *sort* attribute in the previous section.

```
{  
  "name": "myJSmodule",  
  "description": "this is an example of a Devcon Javascript module",  
  "visible": true,  
  "sort": -1,  
  "commands": [  
    {"  
      "name": "myFirstCommand",  
      "path": "myFirstCommand.js",  
      "description": "this is my first js command",  
      "context": "NONE",  
      "proxyParams": false,  
      ...  
    }]  
}
```

## Parameter definition

As part of the `command` object in the `commands.json` file we can define the parameters using the following structure of attributes:

- **parameters** array to group the command parameters. For each parameter we will define the following attributes:
  - *name*: the name for the parameter.
  - *description*: the description of the parameter to be shown to the users.

- *optional*: a *true/false* attribute to define if the parameter is mandatory or not.
- *sort*: see the *sort* attribute in the previous section.
- *inputType*: by default the parameters will be represented in the Devcon's graphic user interface as text boxes but, in case we want the parameter to be a drop down list, a directory picker or a password box, we can specify it using this *inputType* attribute and defining some sub-attributes
  - *drop down list*: "inputType": { "name": "list", "values": [ "optionA", "optionB", "optionC" ] }
  - *directory picker*: "inputType": { "name": "path", "values": [ ] }
  - *password box*: "inputType": { "name": "password", "values": [ ] }

In our example we are going to add two parameters, a first one that will be showed as a text box and the second one that will be a drop down with four options. The result will look like the following

```
{
  "name": "myJSmodule",
  "description": "this is an example of a Devcon Javascript module",
  "visible": true,
  "sort": -1,
  "commands": [
    {
      "name": "myFirstCommand",
      "path": "myFirstCommand.js",
      "description": "this is my first js command",
      "context": "NONE",
      "proxyParams": false,
      "parameters": [
        {
          "name": "firstParameter",
          "description": "this is my first parameter",
          "optional": false,
          "sort": -1
        },
        {
          "name": "secondParameter",
          "description": "this is my second parameter",
          "optional": true,
          "sort": -1,
          "inputType": { "name": "list", "values": [ "devonfw", "oasp4j", "cobigen", "devcon" ] }
        }
      ],
      ...
    }
  ],
}
```

## The commands

Each command will be defined in a separate Javascript file with a name that match the *path* attribute defined in the *commands.json* file of the module. Remember that in case that the js file is in the same directory than the *commands.json* file we only need to provide the name of the js file.

The JavaScript file must have as content either a named or anonymous function which contains the command implementation. The parameters of the function contain the parameters in the defined order and the *this* special property points to the Java *CommandModule* context.

So returning to our example we will have a file called `myFirstCommand.js` located in the same directory than the `commands.json`.

The content will be

```
function (firstParameter, secondParameter){  
    // Here the content of your module.  
}
```

[[devcon-command-developers-guide\_creating-a javascript module]] ===== Creating a javascript module

[[devcon-command-developers-guide\_adding-the module structure]] ===== Adding the module structure

We have already seen the structure of a Devcon's Javascript module so let's see how to create one with an example that contains all steps.

In this case we are going to create (again) a command to extract a zip file, so we are going to create a module called `myJModule` with a command `extract` that gets two mandatory parameters `filepath` for the path to the zip file and a `targetpath` to define the location of the extracted files.

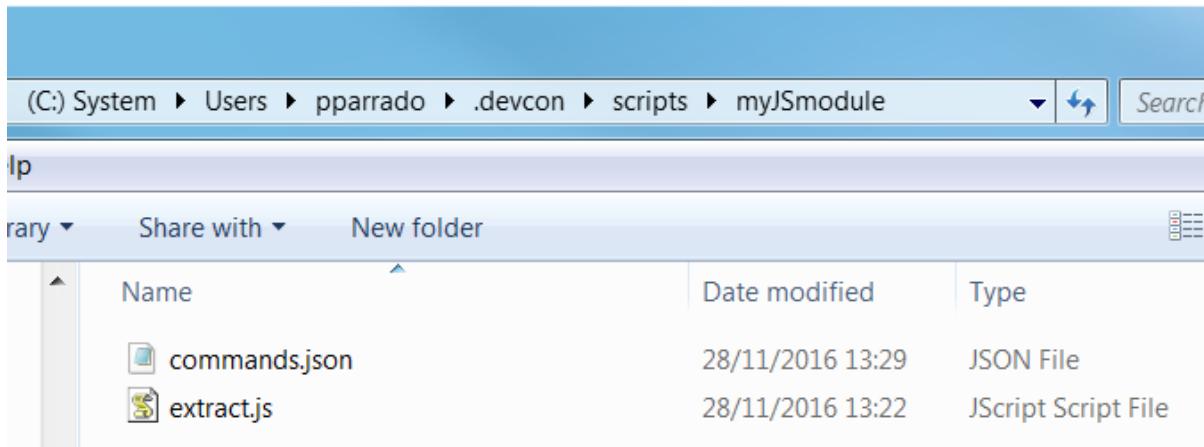
- 1. The Devcon Directory
  - for local installations of Devcon: `C:\Users\{Your User}\.devcon` (if you don't find the `.devcon` directory there try looking in `D:` drive, if the directory is not there neither check your Devcon's installation).
  - for the Devcon tool within the Devon distribution (version 2.1.1 or higher): `Devon-dist\software\devon`

#### Note

If you want to customize a copy of Devcon in a local context and you still don't have Devcon installed you can see how to download and install it [here](#).

- 2. We will need to create the `scripts` folder within the *Devcon Directory*.
- 3. Then we will need to create inside the `scripts` folder the directory for our new module and inside it we need to add
  - a `commands.json` file with the definition of the module
  - and an `extract.js` file with the code for the `extract` command.

So we will end having a structure like `{Devcon Directory}\scripts\myModule`



[[devcon-command-developers-guide\_defining-the module and the command]] ===== Defining the module and the command

To define and register the module and the command we will use the *commands.json* file. First we will add the module metadata (name, description) and then the commands, and its parameters, inside the *commands* array.

```
{
  "name": "myJSmodule",
  "description": "test module",
  "visible": true,
  "sort": -1,
  "commands": [
    {
      "name": "extract",
      "path": "extract.js",
      "description": "command to extract a file",
      "context": "NONE",
      "proxyParams": false,
      "parameters": [
        {
          "name": "filepath",
          "description": "path to the file to be extracted",
          "optional": false,
          "sort": -1
        },
        {
          "name": "targetpath",
          "description": "path to the folder to locate the extracted files",
          "optional": false,
          "sort": -1
        }
      ]
    }
  ]
}
```

[[devcon-command-developers-guide\_adding-the command logic]] ===== Adding the command logic

As we have previously mentioned we need to add the code of our command in the *extract.js* file. As we want to extract a file, to avoid a most complicated implementation, we are going to use the *unZip* method that belongs to the *utils* package of Devcon. To access to the method we will need to provide the fully qualified name `com.devonfw.devcon.common.utils.Extractor.unZip`.

So in the *extract.js* file we must add a function that gets the two parameters defined in the *commands.json* (*filepath* and *targetpath*) and uses the Java method *unZip* to extract the file. Also

remember that the special property `this` will give us access to the Devcon's module context so we will be able to use the Devcon's output (you can find the entire resources that `this` can provide [here](#))

```
function(filepath, targetpath){  
    this.getOutput().showMessage("extracting...");  
    com.devonfw.devcon.common.utils.Extractor.unZip(filepath, targetpath);  
    this.getOutput().showMessage("Done!");  
}
```

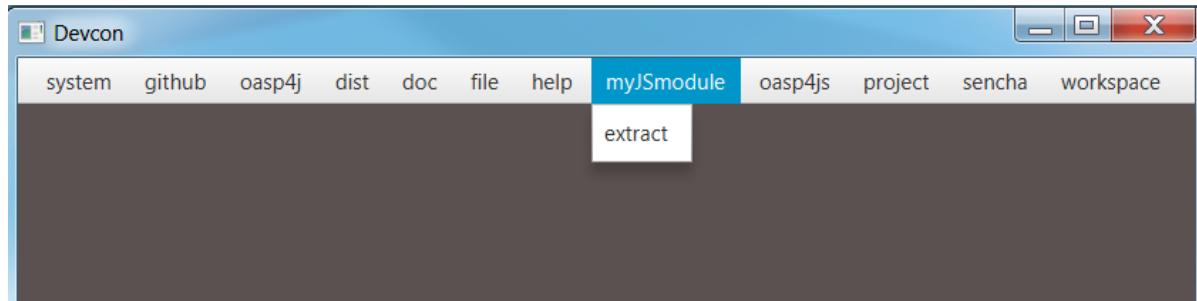
[[devcon-command-developers-guide\_using-the new module and the command]] ===== Using the new module and the command

We have finished the implementation of the new Javascript module so now we can start using it.

We have created a module to extract `zip` files so we are going to use a `myFile.zip` located in the `D:` drive and we are going to extract it to the `D:\Temp` directory using our new module.

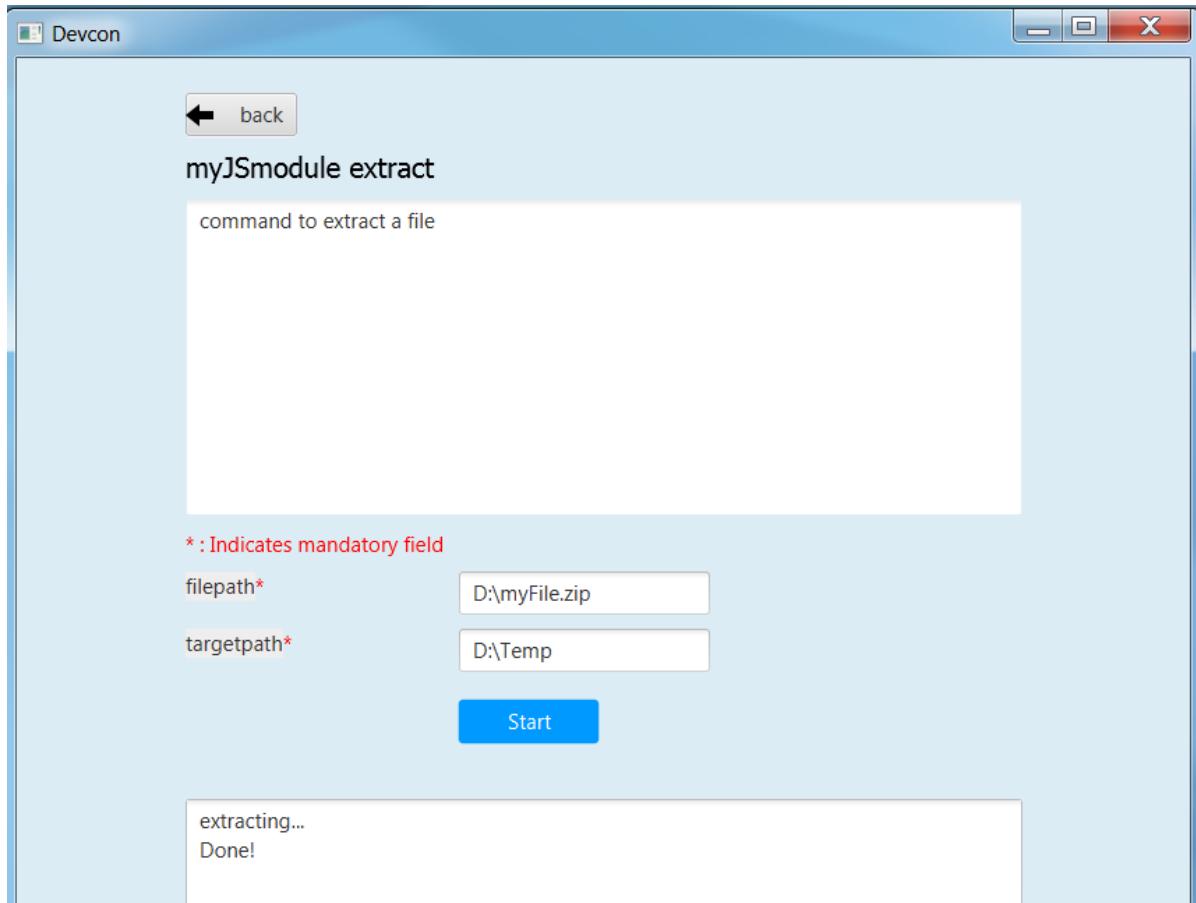
As you may know if you have followed the Devcon's documentation we can use the tool in two ways: using the command line or using the Devcon's graphic user interface (GUI).

[[devcon-command-developers-guide\_using-the gui]] ===== using the gui To launch Devcon's GUI we must open a command line and use the `devon -g` command. After that the Devcon main window should be opened and we should see our new `myJSModule` in the list of available modules. Then if we click over the module we should see the `extract` command available.



Then if we click over the `extract` command we should see a window with the name and description we provided in the `commands.json` alongside the parameters that we defined (`filepath` and `targetpath`), both mandatory.

If we provide the parameters and click on the `Start` button the command should be launched and the file should be extracted.



We have extracted the file successfully using our just created Devcon command.

[[devcon-command-developers-guide\_using-the command line]] ===== using the command line If we use the command line the result will be exactly the same.

Open a Windows command line (for local Devcon installations) or *command.line* script (for the Devcon included in Devon distributions) and launch the devcon command (devon or devcon -h will also work)

```
...>devon
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: devon <<module>> <<command>> [parameters...] [-g] [-h] [-p] [-s] [-v]
Devcon is a command line tool that provides many automated tasks around the full
life-cycle of Devon applications.
-g,--gui      show devcon GUI
-h,--help     show help info for each module/command
-p,--prompt   prompt user for parameters
-s,--stacktrace show (if relevant) stack-trace when errors occur
-v,--version   show devcon version
List of available modules:
> dist: Module with general tasks related to the distribution itself
> doc: Module with tasks related with obtaining specific documentation
> file: custom devcon module
> github: Module to get Github repositories related to Devonfw.
> help: This module shows help info about devcon
> myJSmodule: test module
> oasp4j: Oasp4j(server project) related commands
> oasp4js: Module to automate tasks related to oasp4js
> project: Module to automate tasks related to the devon projects (server + client)
> sencha: Commands related with Ext JS6/Devon4Sencha projects
> system: Devcon and system-wide commands
> workspace: Module to create a new workspace with all default configuration
```

In the list of available modules you should see our `myJSmodule`.

Now if we ask for the `myJSmodule` information with the command `devcon myJSmodule -h` we can check that our `extract` command is available. Also we can see the needed parameters using the `devcon myJSmodule extract -h` command

```
...>devcon myJSmodule extract -h
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
usage: myJSmodule extract [-filepath] [-targetpath]
command to extract a file
  -filepath    path to the file to be extracted
  -targetpath  path to the folder to locate the extracted files
```

Finally we can use the `extract` command providing both mandatory parameters

```
...>devcon myJSmodule extract -filepath D:\myFile.zip -targetpath D:\Temp
Hello, this is Devcon!
Copyright (c) 2016 Capgemini
extracting...
file unzip : D:\Temp\myFile\file1.txt
file unzip : D:\Temp\myFile\file2.txt
file unzip : D:\Temp\myFile\file3.txt
file unzip : D:\Temp\myFile\file4.txt
Done!
```

## 5.4.2 Conclusion

In this section we have seen how easy can be to extend Devcon with new modules. You can either choose to add a Java module into the core of Devcon or achieve the same in an external way creating your own modules with Javascript (remember that you will need Java 8 to run your Javascript modules).

Thanks to the Devcon's structure, in both cases the work is reduced to, first, register the modules and then define each of its elements (commands and parameters) and the modules engine of Devcon will do the rest.

## 5.5 Devon module Developer's Guide

Here, we intend to explain, creation of a maven module (devonfw module), which will serve as template for creating the other modules. This is done with the usage of maven archetype plugin. Please, follow the subsequent sections, for creation of the same.

[[devon-guide-devon-module-developer-guide\_creation-of module template]] == Creation of module template

The maven Archetype Plugin allows the user to create a Maven project from an existing template called an archetype. It also allows the user to create an archetype from an existing project.

**Example Project**

**archetype:create-from-project**

Creating an archetype from an existing project involves three steps:

- the archetype resolution
- the archetype installation:deployment
- the archetype usage

**1.** Using devon distribution's, *console.bat*, open console and go to directory where project exists, and then execute the command:

```
mvn archetype:create-from-project
```

**2.** It then generates the directory tree of the archetype in the *target/generated-sources/archetype* directory.

**3.** Then, move to the mentioned directory above and execute,

```
mvn install
```

So, this step installs the created archetype in users local maven repository.

### 5.5.1 Usage

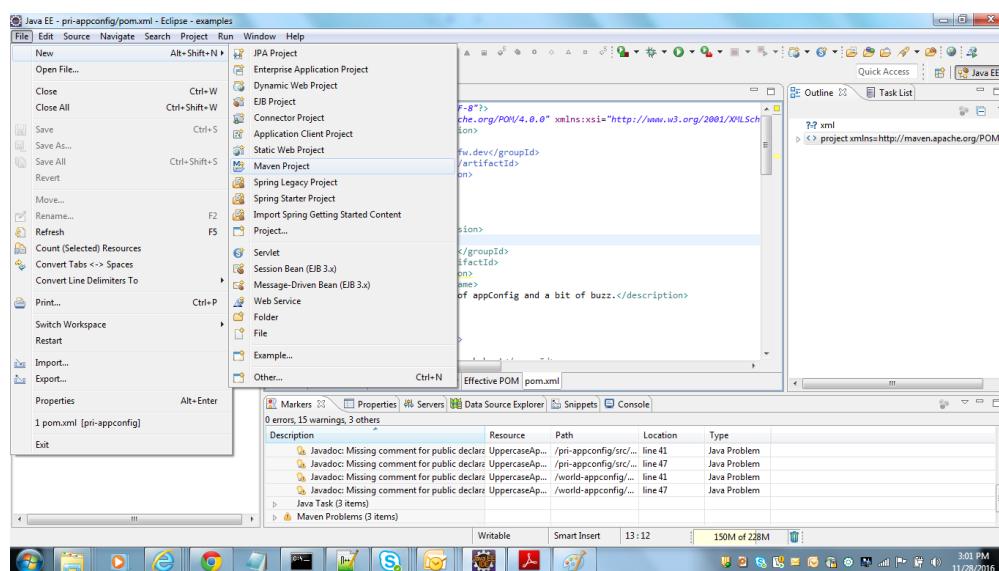
One can use this archetype to create maven modules/projects with:

- Eclipse
- From command line

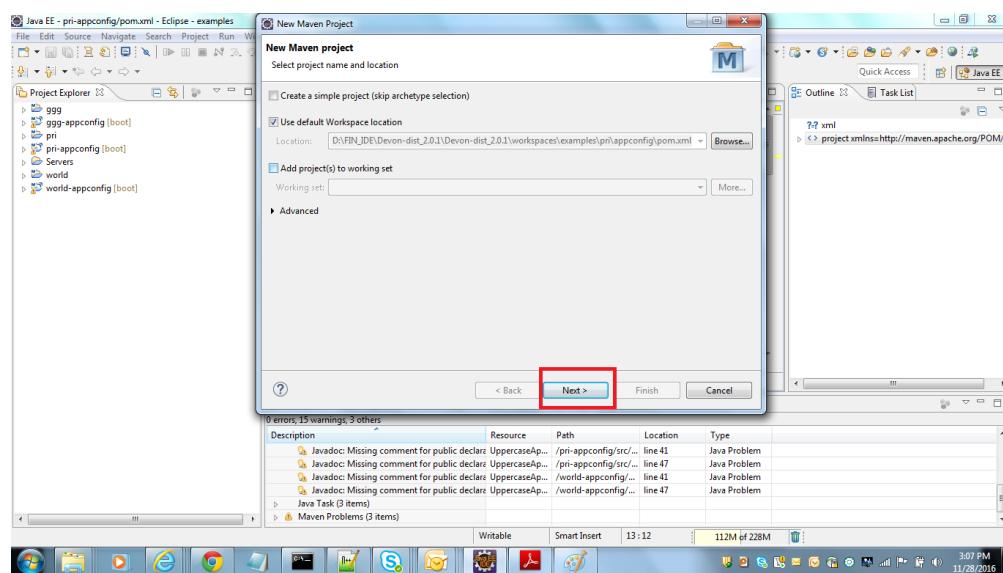
[[devon-guide-devon-module-developer-guide\_creation-of module/project using eclipse]] === Creation of module/project using Eclipse

If one wants to create project using archetype in Eclipse, follow below steps:

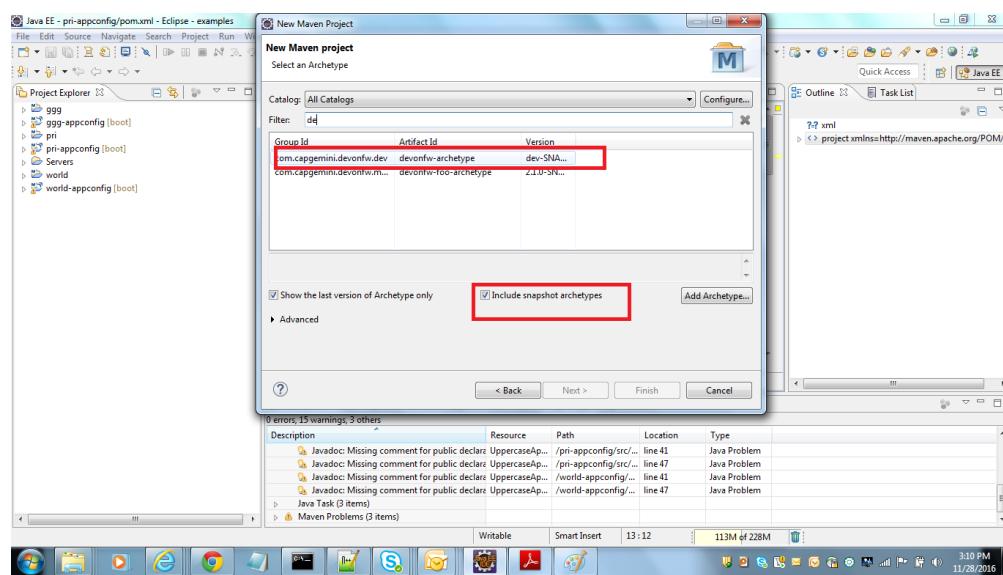
Go to File → New → Maven Project



Click on the Next button.



Select archetype.



For the very first time, when we use archetype in Eclipse, it sometimes does not appear in the list of available archetypes. So in that case, use "add archetype" button.

Once you select archetype, and press "Next" button, a dialog appears, where you need to put desired *artifactid* and *group id* and click finish button. A new project is created on the basis of chosen archetype.

[[devon-guide-devon-module-developer-guide\_creation-of module through command line]] ===  
 Creation of module through command line

To use command line, go to your devon distribution and run *console.bat* Once the console is opened, execute the command:

```
mvn -DarchetypeVersion=dev-SNAPSHOT -DarchetypeGroupId=devonfw-archetype
-DarchetypeArtifactId=com.capgemini.devonfw.dev archetype:generate
-DgroupId=com.devonfw.modules -DartifactId=samplemodule -Dversion=0.1-SNAPSHOT
-Dpackage=com.devonfw.modules.samplemodule
```

As, we are using Eclipse Neon version and it mandates the usage of java 8. So, if you don't have the latest devon distribution and you want to use this archetype for the module creation, follow below steps:

1. When you use archetype with java version lower than java 8, project will be created and you will get error like:

**Unbound classpath container: 'JRE System Library [JavaSE-1.8]' in project {project\_name}**, you will have to manually point installed JRE to desired version. Please, refer [here](#).

1. Once, the project is created and step 1 is done, you need to manually change the java version in *pom.xml* to the desired java version. So, the generated *pom.xml* would have version as shown below:

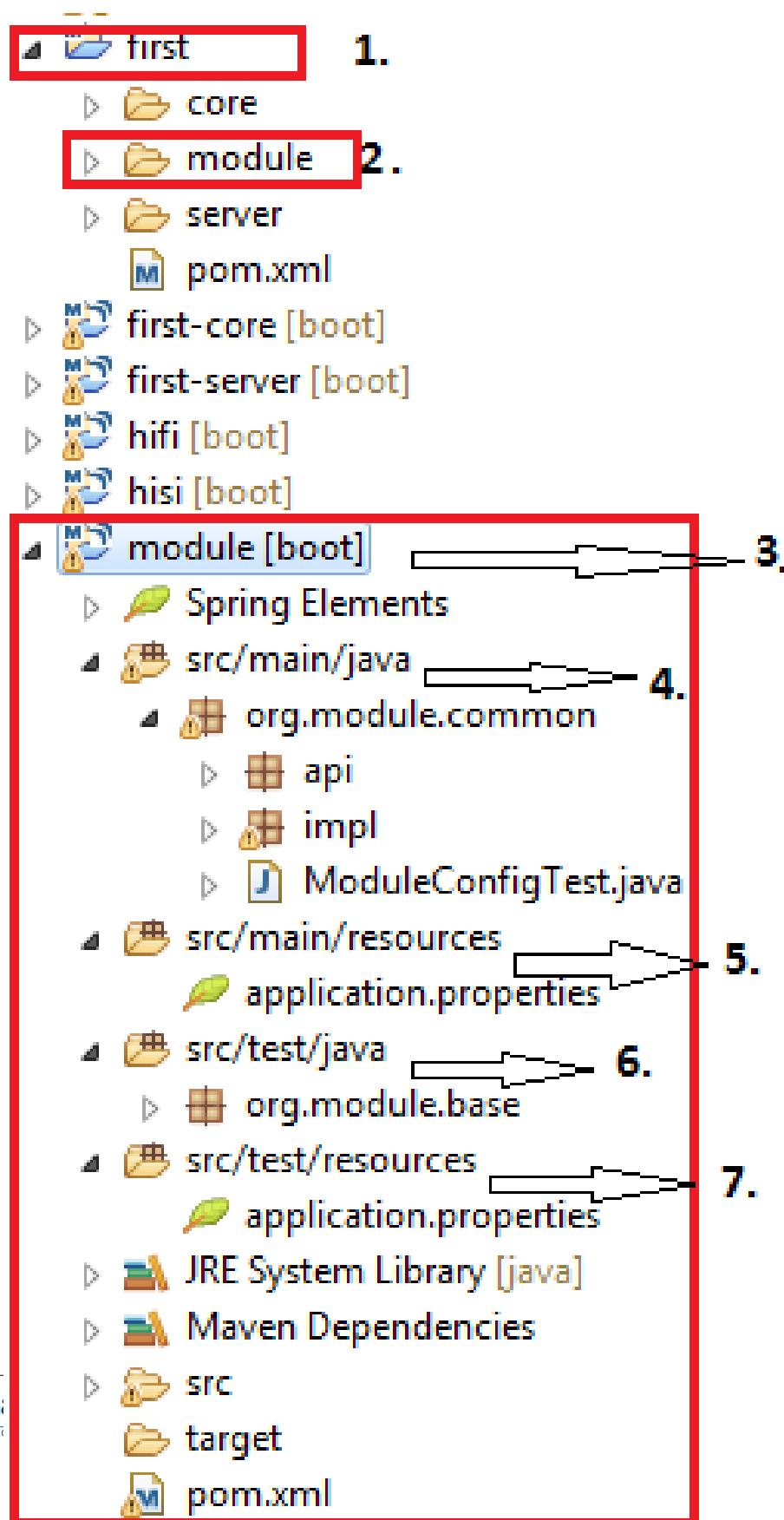
```
<java.version>1.8</java.version>
```

Please, change it to

```
<java.version>1.7</java.version>
```

Once, these 2 steps are done, then you can proceed with newly created project/module.

[[devon-guide-devon-module-developer-guide\_structure-of created module]] == Structure of created module Once, above steps are followed, created module structure would look like below:



---

Here are the details of the structure:

At the top level files descriptive of the project: a *pom.xml* file. In addition, there are textual documents meant for the user to be able to read immediately on receiving the source: README.txt, LICENSE.txt, etc.

There are just two subdirectories of this structure: src and target. The only other directories that would be expected here are metadata like CVS, .git or .svn, and any subprojects in a multiproject build (each of which would be laid out as above).

The target directory is used to house all the output of the build.

The src directory contains all of the source material for building the project, its site and so on. It contains a subdirectory for each type: main for the main build artifact, test for the unit test code and resources, site and so on.

1. "first" is the maven multi module project. Lets say, you want to create a new module in this project using new module archetype.
2. "module" created using archetype.
3. "module[boot]" the whole structure like *src/main/java* etc can be seen inside it.
4. Directory for the language java.
5. contains all the resources like .properties file etc. For example, *application.properties*.
6. Contains all test classes(.java), like junit etc.
7. Contains resources required for testing purposes.

## 5.6 Steps to use Devon distribution in Linux OS

- Navigate to devon dist folder and open a terminal.
- On the terminal, execute env.sh script, using [ . ]. [ . ] is a command in linux shell , it sources a script , so it is included into your current shell environment.

```
. env.sh
```

- Executing of this script, guarantees that all environment variables like JAVA\_HOME, M2\_HOME are set.
- Once above steps are done, run create-or-update-workspace script, which inturn creates scripts for eclipse like eclipse-main. And then , you can have things or steps which you want. Above steps are necessary for Linux distribution, so that all environment variables are set. For more details on devon distro in general, refer [here](#).

## 5.7 List of Components

[[components-list\_devonfw-release 2.2.0]] == Devonfw Release 2.2.0

Table 5.1. Environment Elements

Element	Version
ant	1.9.6
Eclipse	4.6 (Neon)
jasypt	1.9.2
java	1.8.0_101
maven	3.3.9
nodejs	6.11.0
sencha	6.2.1.29
Tomcat	8.0.32
subversion	1.9.3

Table 5.2. Devon Components

Component	Version
OASP4J	2.4.0
devon4sencha	2.0.1
Oasp4js	---

Table 5.3. IP Modules

IP Module	Version	Description
Devcon	1.3.0	Devonfw UI / Task runner / automation tool
Cobigen (plugin Eclipse)	2.1.1	Code generator UI in Eclipse
Cobigen core	3.0	Code generator
Reporting	2.2.0	Reporting module for OASP4J based on JasperReports
Winauth-ad	2.2.0	App authentication against Active Directory
Winauth-sso	2.2.0	App authentication with Windows credentials
<b>Compose for Redis</b>	<b>2.2.0</b>	<b>Implementation of Compose for Redis</b>

IP Module	Version	Description
Async	2.2.0	Implementation for asynchronous web request
i18N	2.2.0	Internationalization
Integration	2.2.0	JMS Implementation
Microservices	2.2.0	Archetypes for Spring Cloud infrastructure
<b>Compose for Redis</b>	<b>2.2.0</b>	<b>Implementation of Compose for Redis</b>

Table 5.4. Java Libraries

Library	Version
javax.transaction API	1.2
mmm-util-search	7.3.0
Querydsl - JPA support	3.4.3
Querydsl - Core module	3.4.3
Guava: Google Core Libraries for Java	17.0.0
Commons Lang	0.2.4
spring-batch-core	3.0.7.RELEASE
com.ibm.jbatch-tck-spi	1.0
javax.batch-api	1.0
XStream Core	1.4.7
xmlpull	1.1.3.1
jettison	1.2
spring-batch-infrastructure	3.0.5.RELEASE
spring-retry	1.1.2.RELEASE
spring-batch-integration	3.0.5.RELEASE
spring-integration-core	4.3.9.RELEASE
oasp4j-logging	2.0.0
Logback Classic Module	1.1.3
Logback Core Module	1.1.3
jcl-over-slf4j	1.7.13
<b>poi</b>	<b>3.14</b>

Library	Version
Javax.inject	1
Janino	2.7.8
Commons-Compiler	2.7.8
HttpComponents Apache HttpClient	4.5.1
Httpcore	4.4.4
Commons-codec	1.9
Oasp4j-beanmapping	2.0.0
Dozer	5.5.1
Commons-beanutils	1.9.2
Commons-lang3	3.2.1
Mmm-util-core	7.0.0
Oasp4j-security	2.0.0
Spring-security-config	4.0.3.RELEASE
Spring-security-core	4.0.3.RELEASE
Spring-security-web	4.0.3.RELEASE
Javax.annotation-api	1.2
Oasp4j-rest	2.0.0
javax.ws.rs-api	2.0.1
Bean Validation API	1.1.0.Final
Spring-context	4.3.8.RELEASE
Expression Language 2.2 Implementation	2.2.4
oasp4j-basic	2.0.0
oasp4j-jpa-envers	2.0.0
hibernate-envers	4.3.11.Final
oasp4j-web	2.0.0
Javax.servlet-api	3.1.0
Spring.aop	4.3.8.RELEASE
Aopalliance	1.0
Spring-beans	4.3.8.RELEASE
<b>poi</b>	<b>3.14</b>

Library	Version
Spring-core	4.3.8.RELEASE
Commons-logging	1.2
Spring-test	4.3.8.RELEASE
Spring-webmvc	4.3.8.RELEASE
Spring-expression	4.3.8.RELEASE
Spring-web	4.3.8.RELEASE
Jstl	1.2
Spring-orm	4.3.8.RELEASE
Spring-jdbc	4.3.8.RELEASE
Spring-tx	4.3.8.RELEASE
Hibernate-entitymanager	4.3.11.Final
Jboss-logging	3.3.0.Final
Jboss-logging-annotations	1.2.0.Beta
Hibernate-core	4.3.11.Final
Antlr	2.7.7
Jandex	1.1.0.Final
Dom4j	1.6.1
Xml-apis	1.0.b.2
Hibernate-commons-annotations	4.0.5.Final
Jboss-transaction-api	1.2
Javassist	3.18.1
H2	1.4.190
Flyway-core	3.2.1
hibernate-jpa-2.1-api	1.0.0.Final
Cglib	3.1
Asm	4.2
Hibernate-validator	5.2.2.Final
Classmate	1.1.0
Cxf-rt-frontend-jaxws	3.1.4
<b>poi</b>	<b>3.14</b>

Library	Version
Xml-resolver	1.2
Cxf-core	3.1.4
Woodstox-core-asl	4.4.1
Stax2-api	3.1.4
Xmllschemacore	2.2.1
Cxf-rt-bindings-soap	3.1.4
Cxf-rt-wsdl	3.1.4
Wsdl4j	1.6.3
Cxf-rt-databinding-jaxb	3.1.4
Jaxb-impl	2.2.11
Jaxb-core	2.2.11
Cxf-rt-bindings-xml	3.1.4
Cxf-rt-frontend-simple	3.1.4
Cxf-rt-ws-addr	3.1.4
Cxf-rt-ws-policy	3.1.4
Neethi	3.0.3
Cxf-rt-frontend-jaxrs	3.1.4
Cxf-rt-rs-service-description	3.1.4
Cxf-rt-transports-http	3.1.4
Jackson-jaxrs-json-provider	2.4.2
Jackson-jaxrs-base	2.4.2
Jackson-core	2.6.3
Jackson-module-jaxb-annotations	2.4.2
Spring-websocket	4.3.8.RELEASE
Spring-messaging	4.3.8.RELEASE
Spring-batch-test	3.0.5.RELEASE
Commons-collections	3.2..1
Commons-io	2.4
Hamcrest-all	1.3
<b>poi</b>	<b>3.14</b>

Library	Version
Oasp4j-test	2.0.0
Assertj-core	2.0.0
Memoryfilesystem	0.6.4
Mockito-core	1.10.19
Objenesis	2.1
Javax.el-api	2.2.4
Spring-boot-starter-web	1.5.3.RELEASE
Spring-boot-starter	1.5.3.RELEASE
Spring-boot	1.5.3.RELEASE
Spring-boot-autoconfigure	1.5.3.RELEASE
Spring-boot-starter-logging	1.5.3.RELEASE
Jul-to-slf4j	1.7.13
Log4j-over-slf4j	1.7.13
Snakeyaml	1.16
Spring-boot-starter-tomcat	1.5.3.RELEASE
Tomcat-embed-core	8.0.28
Tomcat-embed-el	8.0.28
Tomcat-embed-logging-juli	8.0.28
Tomcat-embed-websocket	8.0.28
Spring-boot-starter-validation	1.5.3.RELEASE
Spring-boot-starter-jdbc	1.5.3.RELEASE
Tomcat-jdbc	8.0.28
Tomcat-juli	8.0.28
Spring-boot-starter-actuator	1.5.3.RELEASE
Spring-boot-actuator	1.5.3.RELEASE
Spring-boot-starter-security	1.5.3.RELEASE
Scala-library	2.10.4
Spring-boot-starter-ws	1.5.3.RELEASE
Spring-jms	4.3.8.RELEASE
<b>poi</b>	<b>3.14</b>

Library	Version
Spring-oxm	4.3.8.RELEASE
Spring-ws-core	2.2.3.RELEASE
Spring-xml	2.2.3.RELEASE
Spring-ws-support	2.2.3.RELEASE
Junit	4.12
Hamcrest-core	1.3
Slf4j-api	1.7.13
jgit	4.4.0.201605250940-rc1
jsch	0.1.53
javaEWAH	0.7.9
reflections	0.9.10
javaassist	3.19.0
annotations	2.0.1
commons-cli	1.2
commons-lang3	3.4
commons-exec	1.3
json-simple	1.1.1
commons-io	2.5
java-semver	0.9.0
javax.activation	1.0.2
axis	1.4
commons-discovery	0.2
commons-logging	1.0.4
JAX-RPC	1.1
javax.mail	1.3.1
javax/xml/soap (SAAJ)	1.2
httpclient	4.3.6
httpcore	4.3.3
commons-codec	1.6
<b>poi</b>	<b>3.14</b>

Library	Version
httpasyncclient	4.0.2
httpmime	4.3.6
jasperreports	6.2.1
itext	2.1.7.js5
<b>poi</b>	<b>3.14</b>

## 5.8 Devonfw Release notes 2.2 "Courage"

[[release-notes\_production-line integration]] == Production Line Integration Devonfw is now fully supported on the Production Line v1.3 and the coming v2.0. Besides that, we now "eat our own dogfood" as the whole devonfw project, all "buildable assets", now run on the Production Line.

### 5.8.1 OASP4js 2.0

The main focus of the Courage release is the renewed introduction of "OASP for JavaScript", or OASP4js. This new version is a completely new implementation based on Angular (version 4). This new "stack" comes with:

- New application templates for Angular 4 application (as well as Ionic 3)
- A new reference application
- A new tutorial (and Architecture Guide following soon)
- Component Gallery
- New Cobigen templates for generation of both Angular 4 and Ionic 3 UI components ("screens")
- Integration of Covalent and Bootstrap offering a large number of components
- my-thai-star, a showcase and reference implementation in Angular of a real, responsive usable app using recommended architecture and patterns
- A new Tutorial using my-thai-star as a starting point

See: <https://github.com/oasp/oasp4js-application-template> <https://github.com/oasp/oasp4js-angular-catalog> <https://github.com/oasp/my-thai-star/tree/develop/angular>

[[release-notes\_a-new oasp portal]] == A new OASP Portal As part of the new framework(s) we have also done a complete redesign of the OASP Portal website at <http://oasp.io/> which should make all things related with OASP more accessible and easier to find.

[[release-notes\_new-cobigen ]] == New Cobigen

Major changes in this release:

- Support for multi-module projects
- Client UI Generation:
  - New Angular 4 templates based on the latest - angular project seed
  - Basic Typescript Merger
  - Basic Angular Template Merger
  - JSON Merger
- Refactored oasp4j templates to make use of Java template logic feature
- Bugfixes:
  - Fixed merging of nested Java annotations including array values

- more minor issues
- Under the hood:
  - Large refactoring steps towards language agnostic templates formatting sensitive placeholder descriptions automatically formatting camelCase to TrainCase to snake-case, etc.
- Easy setup of CobiGen IDE to enable fluent contribution
- CI integration improved to integrate with GitHub for more valuable feedback

See: <https://github.com/devonfw/tools-cobigen/releases>

[[release-notes\_mythaistar:-new restaurant example, reference implementation & methodology showcase]] == MyThaiStar: New Restaurant Example, reference implementation & Methodology showcase

A major part of the new devonfw release is the incorporation of a new application, "my-thai-star" which among others:

- serve as an example of how to make a "real" devonfw application (i.e. the application could be used for real)
- Serves as an attractive showcase
- Serves as a reference application of devonfw patterns and practices as well as the standard example in the new devonfw tutorial
- highlights modern security option like JWT Integration

The application is accompanied by a substantial new documentation asset, the devonfw methodology, which described in detail the whole lifecycle of the development of a devonfw application, from requirements gathering to technical design. Officially my-thai-star is still considered to be an incubator as especially this last part is still not as mature as it could be. But the example application and tutorial are 100% complete and functional and form a marked improvement over the "old" restaurant example app. My-Thai-star will become the standard example app from devonfw 3.0 onwards.

See: <https://github.com/oasp/my-thai-star> <https://github.com/oasp/my-thai-star/wiki>

[[release-notes\_the-new oasp tutorial]] == The new OASP Tutorial The OASP Tutorial is a new part of the combined OASP / devonfw documentation which changes the focus of how people can get started with the platform

There are tutorials for OASP4j, OASP4js (Angular), OASP4fn and more to come. My-Thai-Star is used throughout the tutorial series to demonstrate the basic principles, architecture, and good practices of the different OASP "stacks". There is an elaborated exercise where the readers get to write their own application "JumpTheQueue".

We hope that the new tutorial offers a better, more efficient way for people to get started with devonfw. Answering especially the question: how to make a devonfw application.

Oasp4j tutorial: <https://github.com/oasp/oasp-tutorial-sources/wiki/OASP4jGettingStartedHome>  
Oasp4js tutorial: <https://github.com/oasp/oasp-tutorial-sources/wiki/OASP4jsGettingStartedHome>  
Oasp4fn tutorial: <https://github.com/oasp/oasp-tutorial-sources/wiki/OASP4FnGettingStartedHome>

## 5.8.2 OASP4j 2.4.0

"OASP for Java" or OASP4j now includes updated versions of the latest stable versions of Spring Boot and the Spring Framework and all related dependencies. This allows guaranteed, stable, execution of any devonfw 2.X application on the latest versions of the Industry Standard Spring stack. Another important new feature is a new testing architecture/infrastructure. All database options are updated to the latest versions as well as guaranteed to function on all Application Servers which should cause less friction and configuration time when starting a new OASP4j project.

Details:

- Spring Boot Upgrade to 1.5.3
- Updated all underlying dependencies
- Spring version is 4.3.8
- Exclude Third Party Libraries that are not needed from sample restaurant application
- Bugfix:Fixed the 'Whitelabel' error received when tried to login to the sample restaurant application that is deployed onto external Tomcat
- Bugfix:Removed the API `api.org.apache.catalina.filters.SetCharacterEncodingFilter` and used spring framework's API `org.springframework.web.filter.CharacterEncodingFilter` instead
- Bugfix:Fixed the error "class file for javax.interceptor.InterceptorBinding not found" received when executing the command 'mvn site' when trying to generate javadoc using Maven javadoc plugin
- Removed `io.oasp.module.web.common.base.PropertiesWebApplicationContextInitializer` the deprecated API
- Documentation of the usage of `UserDetailsService` of Spring Security

See: <https://github.com/oasp/oasp4j>

Wiki: <https://github.com/oasp/oasp4j/wiki>

## 5.8.3 Microservices Netflix

Devonfw now includes a microservices implementation based on Spring Cloud Netflix. It provides a Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment. It offers microservices archetypes and a complete user guide with all the details to start creating microservices with Devonfw.

See: <https://github.com/devonfw/devon/wiki/devon-microservices>

[[release-notes\_devonfw-distribution based on eclipse oomph]] == Devonfw distribution based on Eclipse OOMPH The new Eclipse devonfw distribution is now based on Eclipse OOMPH, which allows us, in any engagement, to create and manage the distribution more effectively by formalizing the setup instructions so they can be performed automatically (due to a blocking issue postponed to devonfw 2.2.1 which will be released a few weeks after 2.2.0)

[[release-notes\_visual-studio code / atom]] == Visual Studio Code / Atom The devonfw distro now contains Visual Studio Code alongside Eclipse in order to provide a default, state of the art, environment for web based development.

See: <https://github.com/oasp/oasp-vscode-ide>

[[release-notes\_more-i18n options]] == More I18N options The platform now contains more documentation and a conversion utility which makes it easier to share i18n resource files between the different frameworks.

See: <https://github.com/devonfw/devon/wiki/cookbook-i18n-resource-converter>

[[release-notes\_spring-integration as devonfw module]] == Spring Integration as devonfw Module This release includes a new module based on the Java Message Service (JMS) and Spring Integration which provides a communication system (sender/subscriber) out-of-the-box with simple channels (only to send and read messages), request and reply channels (to send messages and responses) and request & reply asynchronously channels.

See: <https://github.com/devonfw/devon/wiki/cookbook-integration-module>

[[release-notes\_devonfw-harvest contributions]] == Devonfw Harvest contributions Devonfw contains a whole series of new components obtained through the Harvesting process. Examples are :

- New backend IP module Compose for Redis: management component for cloud environments. Redis is an open-source, blazingly fast, key/value low maintenance store. Compose's platform gives you a configuration pre-tuned for high availability and locked down with additional security features. The component will manage the service connection and the main methods to manage the key/values on the storage. The library used is "lettuce".
- Sencha component for extending GMapPanel with the following functionality :
  - Markers management
  - Google Maps options management
  - Geoposition management
  - Search address and coordinates management
  - Map events management
  - Map life cycle and behavior management
- Sencha responsive Footer that moves from horizontal to vertical layout depending on the screen resolution or the device type. It is a simple functionality but we consider it very useful and reusable.

See: <https://github.com/devonfw/devon/wiki/cookbook-compose-for-redis-module>

[[release-notes\_more-deployment options to jee application servers and docker/cloudfoundry]] == More Deployment options to JEE Application Servers and Docker/CloudFoundry

The platform now fully supports deployment on the latest version of Weblogic, Websphere, Wildfly (JBoss) as well as Docker and Cloudfoundry

See: <https://github.com/devonfw/devon/wiki/Deployment-on-WebLogic> <https://github.com/devonfw/devon/wiki/cookbook-Deployment-on-WebSphere> <https://github.com/devonfw/devon/wiki/cookbook-Deployment-on-Wildfly>

[[release-notes\_devcon-on linux]] == Devcon on Linux Devcon is now fully supported on Linux which, together with the devonfw distro running on Linux, makes devonfw fully multi-platform and Cloud compatible (as Linux is the default OS in the Cloud!)

See: <https://github.com/devonfw/devcon/releases>

[[release-notes\_new-oasp\_incubators]] == New OASP Incubators From different Business Units (countries) have contributed "incubator" frameworks:

- OASP4NET (Stack based on .NET Core / .NET "Classic" (4.6))
- OASP4X (Stack based on Xamarin)
- OASP4Fn (Stack based on Node.js/Serverless): <https://github.com/oasp/oasp4fn>

An "incubator" status means that the frameworks are production ready, all are actually already used in production, but are still not fully compliant with the OASP definition of a "Minimally Viable Product".

During this summer the OASP4NET and OASP4X repos will be properly installed. In the mean time, if you want to have access to the source code, please contact the *Devonfw Core Team*.