appNG Application Developer Guide

Matthias Müller, Claus Stümke, Matthias Herlitzius

Version 1.18.0-RC1 created on 2018-08-14

Table of Contents

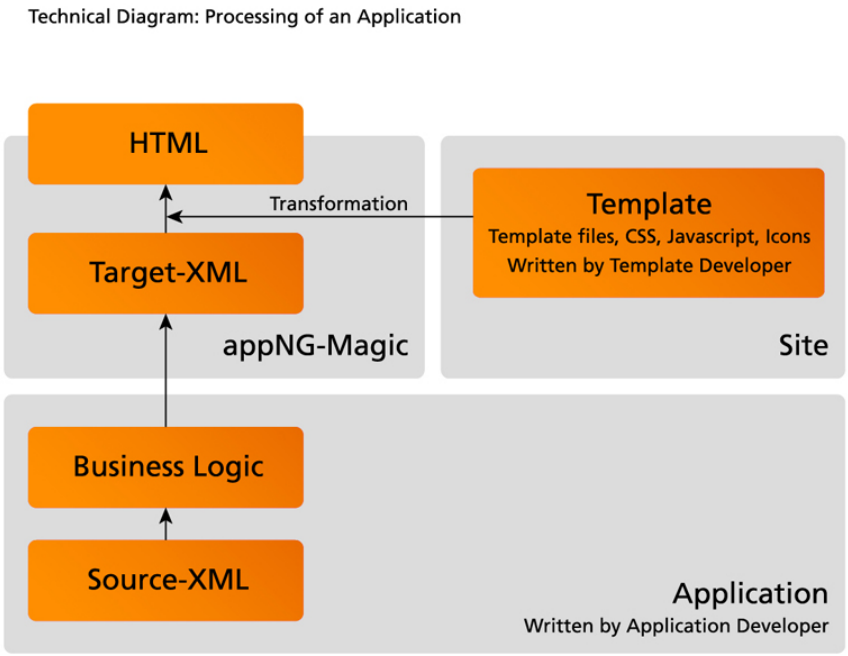
1. Scope

This document describes how to develop applications based on the appNG platform.

2. Core concepts

An appNG application relies on three core concepts: XML, Java and XSLT. The basic idea is that you define the **structure** of your user interface in XML and implement the **business logic** in Java. Finally, XSLT is the part that bridges the gap between the internal XML based structure and the HTML based user interface.

The following diagram illustrates these core concepts:



Below a short explanation of the diagram's elements:

- **Source-XML**
  The XML files defining the structure of the user interface.
  The following XML schema is used for the source-XML:
  http://www.appng.org/schema/platform/appng-platform.xsd
- **Business Logic**
  The implementation of the business logic in Java.
- **Template**
  An appNG Template is a set of resources, used to transform the target-XML into HTML.
  A Template consists of several XSL-stylesheet and all the assets (like JavaScript, CSS, icons) needed to render the HTML. It also contains a `platform.xml` file which is the blue-print for the generated target-XML.
- **Target-XML**
  The target-XML is one single XML document using the template's `platform.xml` as a blue print. This XML document is being enriched with XML fragments that result from combining the static

source-XML with the dynamic business logic.

- **HTML**
  The resulting HTML that is delivered to the user's browser.

3. Structure of an appNG application

An appNG application first and foremost is a standard Apache Maven project. Additionally, it uses an `application-home` folder that contains the non-Java resources.

The listing below gives a rough overview about how an appNG application is composed:

```
├──── application-home
│     ├──── conf
│     │     └──── <XML sources>
│     ├──── dictionary
│     │     └──── <resourcebundles>
│     ├──── resources
│     │     └──── <custom resources>
│     ├──── sql
│     │     └──── <type>
│     │           └──── <DDL Scripts>
│     ├──── xsl
│     │     └──── <XSL stylesheets>
│     ├──── application.xml
│     └──── beans.xml
├──── src
│     └──── main
│         ├──── java
│         └──── resources
│     └──── test
│         ├──── java
│         └──── resources
└──── pom.xml
```

3.1. `application-home/application.xml` (required)

This file describes the basic attributes of an application, like its name, version and the version of appNG it has been built for. Additionally, it defines which roles and permissions are required, which configuration properties exist and which kind of database (if any) is needed to run the application.

The XML schema definition (XSD) for `application.xml` is located here:
http://www.appng.org/schema/application/appng-application.xsd

application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://www.appng.org/schema/application" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/application
http://www.appng.org/schema/application/appng-application.xsd">
    <name>myartifactid</name> ①
    <display-name><![CDATA[myapplication]]></display-name>
    <description><![CDATA[enter description here]]></description>
    <long-description><![CDATA[enter long description here]]></long-description>
    <version>1.0-SNAPSHOT</version>
    <timestamp>20160829-1007</timestamp>
    <appng-version>0.11.0</appng-version>

    <roles> ②
        <role admin-role="true"> ③
            <name>Admin</name>
            <description>an administrator with all permissions</description>
            <permission id="output-format.html" />
            <permission id="output-type.webgui" />
        </role>
    </roles>

    <permissions> ④
        <permission id="output-format.html" />
        <permission id="output-type.webgui" />
    </permissions>

    <properties> ⑤
        <property id="hibernateShowSql">false</property>
        <property id="hibernateFormatSql">false</property>
    </properties>

    <datasources> ⑥
        <datasource type="mysql" />
    </datasources>

</application>
```

① general information (name,version etc.)

② the available roles, referencing permissions

③ a role with attribute `admin-role=true` is automatically added to the default admin group in appNG named *Administrators* on installation

④ the available permissions

⑤ the available properties

⑥ the database type(s) supported by the application

3.2. `application-home/beans.xml` (required)

Since each appNG application get it's own Spring Application Context, a `beans.xml` file must be provided to set up this context.

3.2.1. placeholders

Inside `beans.xml`, you can use several placeholders, as explained below:

- **application properties** as defined in `application.xml`
  Syntax: `${<propertyName>}`
  Example: `${myAppProperty}`
- **site properties** defined for the org.appng.api.model.Site
  Constants for the property names are available in org.appng.api.SiteProperties
  Syntax: `${site.<propertyName>}`
  Example: `${site.mailHost}`
- **platform properties**
  Constants for the property names are available in org.appng.api.Platform.Property
  Syntax: `${platform.<propertyName>}`
  Example: `${platform.devMode}`

Platform properties are only available if the application is configured as a privileged application.

Nested beans and profiles

You can use nested `<beans>` inside `beans.xml`, which can nicely be combined using the `profile` -attribute and the application property activeProfiles.

3.3. `application-home/conf` (optional)

This folder contains the XML definitions of the application's Actions, Datasources and Pages. This folder can have any number of subfolders.

3.4. `application-home/dictionary` (optional)

This folder contains the resource bundle files used for internationalization.

3.5. `application-home/lib` (required)

This folder contains the application's JAR file as well as 3rd party JARs that are not provided by the appNG platform.

3.6. `application-home/sql` (optional)

For every `<datasource>` listed in application.xml, there must be a subfolder containing the DDL

scripts for that type. Since appNG uses Flyway (https://flywaydb.org) to keep the database up to date, the naming scheme `V<version>__<name>.sql` must be used for those scripts, for example `V1.2.5__add_new_table.sql`.

3.7. `application-home/resources` (optional)

This is the place for custom CSS-, Javascript- and image-files, as well as other resources needed by your application. These assets are most likely used by some Custom XSL stylesheets.

3.8. `application-home/xsl` (optional)

This optional folder can contain the custom XSL stylesheets of the application. See Custom XSL stylesheets for more details on that topic.

3.9. `pom.xml` (required)

The `pom.xml` makes use of the parent pom `appng-application-parent` and configures the Maven Assembly Plugin.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>mygroupid</groupId>
    <artifactId>myartifactid</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>myapp</name>
    <description>enter description here</description>

    <parent>
        <groupId>org.appng</groupId>
        <artifactId>appng-application-parent</artifactId>
        <version>0.11.0</version>
    </parent>

    <properties>
        <projectId>${project.name}</projectId>
        <displayName>myapplication</displayName>
        <longDescription>enter long description here</longDescription>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.appng</groupId>
            <artifactId>appng-testsupport</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

The dependency tree of a standard appNG application looks like this:

```
org.appng:appng-api:jar:1.18.0-RC1:provided
  +- org.appng:appng-forms:jar:1.18.0-RC1:provided
  |  +- commons-fileupload:commons-fileupload:jar:1.3.3:provided
  |  +- org.slf4j:slf4j-log4j12:jar:1.7.25:provided
  |  |  \- log4j:log4j:jar:1.2.17:provided
  |  +- org.owasp.esapi:esapi:jar:2.1.0.1:provided
  |  |  +- commons-configuration:commons-configuration:jar:1.10:provided
  |  |  |  \- commons-lang:commons-lang:jar:2.6:provided
  |  |  +- commons-beanutils:commons-beanutils-core:jar:1.8.3:provided
  |  |  +- xom:xom:jar:1.2.5:provided
  |  |  +- org.beanshell:bsh-core:jar:2.0b4:provided
  |  |  +- org.owasp.antisamy:antisamy:jar:1.5.3:provided
  |  |  |  +- net.sourceforge.nekohtml:nekohtml:jar:1.9.16:provided
  |  |  |  \- commons-httpclient:commons-httpclient:jar:3.1:provided
  |  |  \- org.apache.xmlgraphics:batik-css:jar:1.8:provided
  |  |     +- org.apache.xmlgraphics:batik-ext:jar:1.8:provided
  |  |     +- org.apache.xmlgraphics:batik-util:jar:1.8:provided
  |  |     \- xml-apis:xml-apis-ext:jar:1.3.04:provided
  |  \- org.jsoup:jsoup:jar:1.11.1:provided
  +- org.appng:appng-xmlapi:jar:1.18.0-RC1:provided
  |  +- org.apache.commons:commons-lang3:jar:3.6:provided
  |  \- net.sf.saxon:Saxon-HE:jar:9.6.0-6:provided
  +- org.appng:appng-tools:jar:1.18.0-RC1:provided
  |  \- net.sf.jmimemagic:jmimemagic:jar:0.1.5:provided
  +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
  +- org.springframework:spring-context:jar:4.3.12.RELEASE:provided
  |  +- org.springframework:spring-aop:jar:4.3.12.RELEASE:provided
  |  +- org.springframework:spring-beans:jar:4.3.12.RELEASE:provided
  |  +- org.springframework:spring-core:jar:4.3.12.RELEASE:provided
  |  \- org.springframework:spring-expression:jar:4.3.12.RELEASE:provided
  +- org.springframework.ws:spring-ws-core:jar:2.4.0.RELEASE:provided
  |  +- commons-logging:commons-logging:jar:1.2:provided
  |  +- org.springframework.ws:spring-xml:jar:2.4.0.RELEASE:provided
  |  +- org.springframework:spring-oxm:jar:4.3.12.RELEASE:provided
  |  +- org.springframework:spring-web:jar:4.3.12.RELEASE:provided
  |  \- org.springframework:spring-webmvc:jar:4.3.12.RELEASE:provided
  +- wsdl4j:wsdl4j:jar:1.6.3:provided
  +- javax.validation:validation-api:jar:2.0.0.Final:provided
  +- commons-io:commons-io:jar:2.6:provided
  +- org.springframework.data:spring-data-commons:jar:1.13.8.RELEASE:provided
  |  +- org.slf4j:slf4j-api:jar:1.7.25:provided
  |  \- org.slf4j:jcl-over-slf4j:jar:1.7.25:provided
  +- com.fasterxml.jackson.core:jackson-core:jar:2.9.2:provided
  +- com.fasterxml.jackson.core:jackson-databind:jar:2.9.2:provided
  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.9.2:provided
  \- joda-time:joda-time:jar:2.9.9:provided
```

4. The Application

4.1. Output-format and -type

The output-format determines the basic format of the output, such as HTML, PDF or XML. The default output-format is *html*. A format can provide several output-types, each of them offering a different layout.

An output-type offers a certain layout for a output-format. For example, there could be one output-type optimized for desktop clients and one optimized for mobile clients. The default output-type for the format *html* is *webgui*. That's the reason why every application that provides a GUI must add the built-in permissions `output-format.html` and `output-type.webgui` to each role that should have access to that GUI.

See chapter 9 for more information about permissions.

4.2. Pre-defined Beans

The Spring application context of an appNG application comes with a number of pre defined beans of different scopes, as listed here:

- `conversionService`
  Type: org.springframework.core.convert.ConversionService
  Scope: singleton

- `messageSource`
  This mesage source is built from the resource bundle file(s) provided in the application's `dictionary` folder.
  Type: org.appng.api.support.ResourceBundleMessageSource
  Scope: singleton

- `datasource`
  Only available if there is a `<datasource>` present in `application.xml`. Usually used to configure a JPA `EntityManagerFactory`, as described in this section.
  Type: javax.sql.DataSource.html
  Scope: singleton

- `environment`
  The `environment` is used to retrieve and set attributes of different Scopes.
  Type: org.appng.api.Environment
  Scope: request

- `request`
  A wrapper for the current javax.servlet.http/HttpServletRequest.html, offering additional framework methods.
  Type: org.appng.api.Request
  Scope: request

- `selectionFactory`

A factory for building selections, see here for more on that topic.
Type: org.appng.api.support.SelectionFactory
Scope: singleton

4.3. Configurable Application Features

There are some built-in features an application can use by utilizing pre-defined application properties in `application.xml`. See below for a list of these properties:

- `featureIndexing (boolean)`
  If set to `true`, the application can add documents to the site's search index. See the chapter about indexing and searching for details.
- `featureImageProcessing (boolean)`
  If set to `true`, the application can obtain a pre configured org.appng.tools.image.ImageProcessor. See Application#getFeatureProvider() and FeatureProvider#getImageProcessor(File sourceFile, String targetFile) for details.
- `activeProfiles (String)`
  A comma-separated list of active profiles, those can be used in `<beans profile="…">` of `beans.xml`.

4.4. Internationalization (I18n)

Supporting different languages is a frequent requirement for many applications. Therefore, different resource bundles can be provided in the `application-home/dictionary`-folder. Any XML element that binds to a org.appng.xml.platform.Label can make use of the built in internationalization capabilities:

- `<action><config><title>`
- `<action><config><description>`
- `<datasource><config><title>`
- `<datasource><config><description>`
- `<applicationConfig><config><title>`
- `<applicationConfig><config><description>`
- `<page><config><title>`
- `<page><config><description>`
- `<event><config><title>`
- `<event><config><description>`
- `<link><label>`
- `<link><confirmation>`
- `<section><title>`
- `<element><title>`
- `<field><label>`
- `<selection><title>`
- `<optionGroup><title>`

The `id` of a `Label` is used as the resource bundle key. It can be parametrized using the `params` -attribute. This attribute allows

- fixed terms, surrounded by single quotes
- (`<page>`-/`<action>`-/`<datasource>`-) parameter expressions using the syntax `${<param-name>}`
- field references using the syntax `#{<field-name>}`

Consider the following label:

```
<label id="item.delete.confirm" params="'ID', ${current.id}" />
```

and the corresponding entry for `item.delete.confirm` from the resource bundle:

```
item.delete.confirm=Do you really want to delete the item with {0}: {1}?
```

At runtime, the `<label>` would be resolved to

```
<label id="item.delete.confirm">Do you really want to delete the item with ID:
4711?</label>
```

4.4.1. The implicit `i18n` variable

In order to make it possible to retrieve values from the dictionary through an expression, a variable named `i18n` of type org.appng.api.support.I18n is added to the expression evaluation context. This is especially useful if different date- and number-formats should be used for different locales. Check out the description of the format-attribute here to see an example.

4.5. Using JPA

An application can easily make use of Spring Data JPA for implementing the data access layer. See the Reference Documentation for more details.

If your application want's to use JPA, follow these simple steps:

1. Add the following dependency to your `pom.xml` (the version is inherited from the parent pom):

   ```
   <dependency>
       <groupId>org.appng</groupId>
        <artifactId>appng-persistence</artifactId>
   </dependency>
   ```

2. Annotate your persistent domain objects with the required JPA annotations:

```
@Entity
public class Employee {
    ....

    @Id
    public Integer getId(){
        return id;
    }
```

3. Add a `<datasource>` to your application.xml:

```
<datasources>
    <datasource type="mysql" />
</datasources>
```

4. For the chosen database type, provide the DDL scripts:

```
application-home
└──── sql
    └──── mysql
        └──── V1.0.0__init_tables.sql
```

5. Create a org.appng.persistence.repository.SearchRepository for each of your entity classes:

```
public interface EmployeeRepository extends SearchRepository<Employee, Integer> {

}
```

6. In your `beans.xml`, add the following configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <context:component-scan base-package="com.myapp" />
```

```
  <jpa:repositories base-package="com.myapp.repository"
      base-class="org.appng.persistence.repository.SearchRepositoryImpl" /> ①

  <tx:annotation-driven /> ②

  <bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager" /> ③

  ④
  <bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceProviderClass"
      value="org.hibernate.jpa.HibernatePersistenceProvider" />
    <property name="persistenceUnitName" value="myapp"/> ⑤
    <property name="dataSource" ref="datasource" /> ⑥
    <property name="jpaVendorAdapter">
      <bean
          class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties"> ⑦
      <props>
        <prop key="hibernate.show_sql">${hibernateShowSql:true}</prop> ⑧
        <prop key="hibernate.format_sql">${hibernateFormatSql:false}</prop>
        <prop key="hibernate.id.new_generator_mappings">false</prop>
      </props>
    </property>
    <property name="packagesToScan"> ⑨
      <list>
        <value>com.myapp.domain</value>
      </list>
    </property>
  </bean>
  <bean id="entityManager"
        class="org.springframework.orm.jpa.support.SharedEntityManagerBean" />  ⑩
</beans>
```

① Enable Spring Data JPA repositories, defining the `base-package` where your repositories reside. As a `base-class`, use `org.appng.persistence.repository.SearchRepositoryImpl`.

② Enable annotation based transaction management (using org.springframework.transaction.annotation.Transactional).

③ Define a transaction manager.

④ Define an `EntityManagerFactory`.

⑤ Choose a meaningful name for the persistence unit.

⑥ The referenced bean `datasource` of type `javax.sql.DataSource` is provided by the platform.

⑦ Define the JPA properties.

⑧ Use some application-properties, also provide a default value.

⑨ Define the package(s) where your entities reside.

⑩ Define an `javax.persistence.EntityManager`.

7. Inject the repositories into your service class(es) and annotate them with `@org.springframework.transaction.annotation.Transactional`:

```
@org.springframework.stereotype.Service
@org.springframework.transaction.annotation.Transactional
public class EmployeeService {

    private EmployeeRepository employeeRepository;

    @Autowired
    public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    // transactional methods here
}
```

For more details on declarative transaction management in Spring, see chapter 17.5 of the Spring reference documentation.

4.5.1. Working with `SearchRepository` and `SearchQuery<T>`

Often your business logic needs to perform search queries bases on dynamic filter criteria (see Filtering for details about filters). A convenient way to handle this is to use a org.appng.persistence.repository.SearchQuery<T>, which was built exactly for this purpose.

In the following example, employees should be returned

- whose last name contains a certain text
- whose first name starts with a certain text
- who are born after a certain date.

The implementation could look like this:

```
public Page<Employee> searchEmployees(String lastName, String firstName,
    Date bornAfter, Pageable pageable) {
  SearchQuery<Employee> query = employeeRepository.createSearchQuery();
  query.contains("lastName", lastName);
  query.startsWith("firstName", firstName);
  query.greaterThan("dateOfBirth", bornAfter);
  Page<Employee> employees = employeeRepository.search(query, pageable);
  return employees;
}
```

This code is easy to read and thus quite self explaining. But wait, what if some or all of the arguments (except `pageable`) are `null`?

The answer is: Everything is fine and works well. The reason for that is, that all query methods of `SearchQuery<T>` are `null`-safe, meaning the given criteria is being ignored if the argument is `null`. Anyhow, you can use `isNull(String name)` if you explicitly want to check for null.

You can make use of the following criteria methods:

- `equals()` / `notEquals()`
- `isNull()` / `isNotNull()`
- `greaterThan()` / `lessThan()`
- `greaterEquals()` / `lessEquals()`
- `in()` / `notIn()`
- `like()` / `notLike()`
- `startsWith()` / `endsWith()`
- `contains()`

In cases where using criteria methods is not sufficient, you can use SearchRepository.search(String queryString, String entityName, Pageable pageable, Object... params) and pass your custom query string to it.

4.5.2. Adding Auditing with Envers

Adding support for Hibernate Envers can be done in these steps:

1. Add `org.hibernate.envers.Audited` and other Envers annotations to your entities.
2. Let repositories extend `org.appng.persistence.repository.EnversSearchRepository`.
3. Use `org.appng.persistence.repository.EnversSearchRepositoryImpl` as `base-class` of `<jpa:repositories>`.
   You can extend the aforementioned class and override getRevisionEntity() for providing you own revision entity.
4. Provide the DDL scripts for the auditing tables and place them in `application-home/sql/<type>`.

Your repository then offers theses methods, defined by org.springframework.data.repository.history.RevisionRepository

- `Revision<N,T> findLastChangeRevision(ID id)`
- `Revision<N,T> findRevision(ID id, N revisionNumber)`
- `Revisions<N,T> findRevisions(ID id)`
- `Page<Revision<N,T>> findRevisions(ID id, Pageable pageable)`

Also check those methods provided by `SearchRepository`:

- `Collection<T> getHistory(ID id)`
- `T getRevision(ID id, Number revision)`

- `Number getRevisionNumber(ID id)`

4.5.3. Using Querydsl

Adding support for Querydsl can be done in three easy steps:

1. configure QueryDSL in the `pom.xml`, check the Querydsl Reference Guide for details
2. let repositories extend `org.appng.persistence.repository.QueryDslSearchRepository`
3. use `org.appng.persistence.repository.QueryDslSearchRepositoryImpl` as `base-class` of `<jpa:repositories>`

The repository then implements org.springframework.data.querydsl.QueryDslPredicateExecutor

4.6. Custom XSL stylesheets

In every non-trivial application, you will reach the point where the standard rendering of the used template is not sufficient. At that point, you will need to write some custom XSL stylesheets.

There are several points in the source XML documents where you can place a reference to such a custom stylesheet:

- `<page><config>`
- `<action><config>`
- `<event><config>`
- `<datasource><config>`
- `<applicationRootConfig><config>`

As an example, we want to display the following field of an action with a jQuery UI Slider:

```
<field name="rating" type="int">
    <label id="rating" />
</field>
```

When writing custom XSLT, you need to know about the resources the used template ships with. As the standard appNG Template ships with jQuery and jQuery UI, no additional `<script>`-resources need to be imported.

The XSL stylesheet for this would look like shown below:

rating.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns="http://www.appng.org/schema/platform" xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs">①
  <xsl:output method="xhtml" />②

  <xsl:template
      match="action[@id='ratingAction']/config/meta-data/field[@name eq 'rating']" ③
      priority="2" mode="form"> ④
    ⑤
    <xsl:param name="field-htmlid" tunnel="yes" />
    <xsl:param name="field-binding" tunnel="yes" />
    <xsl:param name="field-value" tunnel="yes" />
    <xsl:param name="field-attributes" tunnel="yes" />
    <div class="fieldcontainer">⑥
      <div class="label">⑦
        <label class="text" for="rating"><xsl:value-of select="label/text()"/>
</label>
      </div>
      <div class="field">⑧
        <input type="text" name="{$field-binding}" id="{$field-htmlid}"
            class="text" style="width:80px !important" readonly="readonly" />
        <div id="r_slider" style="margin-top:5px;margin-bottom:10px;width:80px" />
      </div>
      <script>⑨
      $(function() {
        $("#r_slider").slider({
          value: <xsl:value-of select="$field-value"/>, min: 0, max: 5,step: 1,
          slide: function( event, ui ) {
            $("#<xsl:value-of select="$field-htmlid"/>").val(ui.value);
          }
        });
        $("#<xsl:value-of select="$field-htmlid"/>").val($("#r_slider").slider("value
"));
      });
      </script>
    </div>
  </xsl:template>
</xsl:stylesheet>
```

① the root element of the stylesheet, defining the required namespaces

② the output method is `xhtml`

③ the `<xsl:template>` uses an XPath expression to match the required field

④ the template applies for display-mode `form`

⑤ we receive some tunnel parameters from higher prioritized templates

⑥ the `<div>`-container for a field

⑦ the `<div>`-container for a field's label

⑧ the `<div>`-container for the actual `<input>` field

⑨ the Javascript markup to build the slider

Now, the custom stylesheet `rating.xsl` needs to be referenced inside the `ratingAction`:

```
<action id="ratingAction">
  <config>
    <title id="rating.create" />
    <template path="recipe.xsl" /> ①
    ...
  </config>
  ...
</action>
```

① reference the template by the relative path to the `application-home/xsl` folder

For further reference, have a look at the corresponding recommendations from the W3C:

- https://www.w3.org/TR/xslt20
- https://www.w3.org/TR/xpath20
- https://www.w3.org/TR/xpath-functions

4.6.1. Referencing custom resources

If your custom XSL stylesheets need to include custom resources from `application-home/resources`, the schema to build the path for these resources is:

`/template_<application-name>/<relative-resource-path>`

This example assumes that `jquery.colorpicker.js` is located at `application-home/resources/colorpicker`

```
<script src="/template_myapp/colorpicker/jquery.colorpicker.js">
```

5. Datasources

5.1. Metadata and fields

A datasource represent either a single item or a collection of items. It needs to define the kind and structure of the returned data. For getting started, imagine the following Java Class `Employee`:

```
public class Employee {
    private Integer id;
    private String lastName;
    private String firstName;
    private Date dateOfBirth;

    //getters and setters here
}
```

This class can be used by a datasource that represents a collection of items (employees, in that case):

```
<datasource id="employees"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd"> ①
    <config>
        <title id="employees" />
        <params>②
            <param name="selectedId" />
        </params>
        <meta-data bindClass="com.myapp.domain.Employee" ③
                result-selector="${current.id eq selectedId}">④
            <field name="id" type="int">⑤
                <label id="id" />
            </field>
            <field name="lastName" type="text">
                <label id="lastName" />
            </field>
            <field name="firstName" type="text">
                <label id="firstName" />
            </field>
            <field name="dateOfBirth" type="date" format="yyyy-MM-dd">
                <label id="dateOfBirth" />
            </field>
        </meta-data>
    </config>
    <bean id="employees">⑥
        <option name="mode" value="list" />
    </bean>
</datasource>
```

① Each datasource needs an unique id.

② It can be parameterized using a `<params>` element.

③ The `<meta-data>` defines which `bindclass` the datasource binds to.

④ The keyword `current` acts as a loop variable for the collection. The expression given in `result-`

`selector` is used to mark an element as selected.

⑤ A property of the bindclass can be referenced with the `name` attribute of the `<field>`. Any valid **JavaBeans** property can be used here, including nested properties. Depending on the `type` of the field, a `format` pattern can be provided.

⑥ Finally, the name of the Spring bean tells appNG which implementation to call. The `<bean>` can be parameterized using several `<option>` elements.

Now let's check how the implementation of the bean `employees` (see number 6 from above) could look like:

```
import org.appng.api.ActionProvider;
import org.appng.api.DataContainer;
import org.appng.api.DataProvider;
import org.appng.api.Environment;
import org.appng.api.FieldProcessor;
import org.appng.api.Options;
import org.appng.api.Request;
import org.appng.api.model.Application;
import org.appng.api.model.Site;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.data.domain.Page;
import org.springframework.stereotype.Component;

import com.myapp.domain.Employee;
import com.myapp.service.EmployeeService;

@Component
@Scope("request") ①
public class Employees implements DataProvider { ②

    private EmployeeService service;

    @Autowired
    public Employees(EmployeeService service) {
        this.service = service;
    }

    ③
    public DataContainer getData(Site site, Application application, Environment
environment,
            Options options, Request request, FieldProcessor fp) {
        DataContainer dataContainer = new DataContainer(fp);
        String mode = options.getOptionValue("mode", "value");④
        if ("list".equals(mode)) {
            Page<Employee> employees = service.findEmployees(fp.getPageable());
            dataContainer.setPage(employees); ⑤
        } else {
            // do something else
        }
        return dataContainer; ⑥
    }

}
```

① Spring annotations are used to define the bean.

② The interface to implement is org.appng.api.DataProvider.

③ The single method to implement is `getData(…)`.

④ Values of the `<options>` defined in the source XML can be accessed through the

org.appng.api.Options parameter.

⑤ Since appNG uses the Page abstraction of Spring Data, we retrieve a Page object from our service.

⑥ A org.appng.api.DataContainer is being returned.

This was straight forward, wasn't it? A datasource returning a collection of items is used to display those items in tabular form and perform some sorting and filtering on this table.

Next, let's see how to write datasource returning a **single** item.

The datasource:

```
<datasource id="employee"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
        <title id="employee" />
        <params>①
            <param name="id" />
        </params>
        <meta-data bindClass="com.myapp.domain.Employee">②
            <field name="id" type="int" readonly="true">③
                <label id="id" />
            </field>
            <field name="lastName" type="text">
                <label id="lastName" />
            </field>
            <field name="firstName" type="text">
                <label id="firstName" />
            </field>
            <field name="dateOfBirth" type="date" format="yyyy-MM-dd">
                <label id="dateOfBirth" />
            </field>
        </meta-data>
    </config>
    <bean id="employees">④
        <option name="mode" value="single" id="${id}"/>
    </bean>
</datasource>
```

① We need the id of the item.

② Use the same bindclass and fields as before.

③ Since we don't want the id to be changed, mark it as read-only.

④ We use the same Spring bean as before, passing the id parameter through an option attribute, using the syntax `${<param-name>}`.

In the implementing bean, we now just have to add the else-case:

```
else if ("single".equals(mode)) {
    Integer id = request.convert(options.getOptionValue("mode", "id"), Integer.class)
;①
    Employee employee = service.getEmployee(id);②
    dataContainer.setItem(employee);③
}
```

① Convert the option to an `Integer`.

② Retrieve the `Employee` from the `service`.

③ Call `setItem` on the `datacontainer`.

A datasource returning a single item is primarily used by action to perform some operations. See Actions for more details.

5.1.1. Field attributes

- `name` (`string`, required)
  The name of the field. Must be a valid JavaBeans property path.
- `type` (`enum`, required)
  The type of the field. See Field Types and display modes for details.
- `binding` (`string`, optional)
  The name of the HTTP parameter this field binds to. Not needed in most cases. One exception is when using the `binding`-attribute of `<meta-data>`, which acts as a prefix, and this prefix should not be used for this field.
- `readonly` (`boolean expression`, optional)
  Whether or not this is a read-only field. Only relevant in display mode **form**.
  Supports expressions using the syntax `${<param-name>}`.
- `hidden` (`boolean expression`, optional)
  Whether or not this is a hidden field.
  Supports expressions using the syntax `${<param-name>}`.
- `format` (`string expression`, optional)
  Some field types do support additional formatting, see Field Types and display modes for details.
  Supports expressions using the syntax `${<param-name>}`. For example, the implicit i18n-variable can be used to read the date format from a resource bundle:

  ```
  <field name="dateOfBirth" type="date" format="${i18n.message('dateFormat')}" >
  ```

- `displayLength` (`integer`, optional)
  Controls after how many characters the field's value should be truncated.
  Only relevant in display mode **table**.

5.1.2. Conditional fields

A field may contain a `<condition>` that uses all of the datasource's parameters in it's `expression`. Additionally, the `current`-variable can be used. See the section about expressions for more details. Example, assuming a parameter `action` exists:

```
<condition expression="${action eq `edit` and current.id gt 5}" />
```

5.2. Field Types and display modes

As mentioned before, a `<field>` can be of different types. The type of a field determines to which Java types it can be read from/written to and how this field is being rendered by the template. But there is a second factor that affects rendering, the **display mode**. An appNG template must support (at least) the two display modes **form** and **table**.

When a datasource is referenced by (meaning: embedded into) an action, the display mode **form** is used. This means that a field renders as an HTML form element (`<input>`,`<select>` or `<textarea>`). The field attributes `hidden` and `readonly` and also its `<condition>` are evaluated here.

When a datasource is used "standalone" inside a page, the display mode **table** is used. The datasource gets rendered as a HTML `<table>`, using the field labels as table header (`<th>`).

A conclusion from the above is that not every field type is feasible for every display mode. Some field type do support the `format` attribute, which controls how the field's value is being formatted.

See below for a list of all field types.

5.2.1. `text`

Target Java Type: `java.lang.String`

Form Mode: `<input type="text">`

Table Mode: standard string representation

5.2.2. `longtext`

Target Java Type: `java.lang.String`

Form Mode: `<textarea>`

Table Mode: standard string representation

5.2.3. `richtext`

Target Java Type: `java.lang.String`

Form Mode: `<textarea>` with richtext editing capabilities

Table Mode: standard string representation

Supported format: a comma separated list of allowed HTML tags and attributes, e.g `format="p[align],div,span,a[href,target]"` (only relevant in form mode)

5.2.4. `password`

Target Java Type: `java.lang.String`

Form Mode: `<input type="password">`

Table Mode: standard string representation

5.2.5. `url`

Target Java Type: `java.net.URL`

Form Mode: `<input type="text">`

Table Mode: standard string representation

5.2.6. `int`

Target Java Type: `java.lang.Integer`

Form Mode: `<input type="text">`

Table Mode: formatted string representation (standard format: `#`)

Supported format: a DecimalFormat pattern, e.g. `format="#0.00"`,

5.2.7. `long`

Target Java Type: `java.lang.Long`

Form Mode: `<input type="text">`

Table Mode: formatted string representation (standard format: `#`)

Supported format: a DecimalFormat pattern, e.g. `format="#0.00"`

5.2.8. `decimal`

Target Java Type: `java.lang.Float`, `java.lang.Double`

Form Mode: `<input type="text">`

Table Mode: formatted string representation (standard format: `#.##`)

Supported format: a DecimalFormat pattern, e.g. `format="#0.00"`

5.2.9. checkbox

Target Java Type: `java.lang.Boolean`, `boolean` (primitive)

Form Mode: `<input type="checkbox" value="true">`

Table Mode: standard string representation

5.2.10. coordinate

Target Java Type: org.appng.tools.locator.Coordinate

Form Mode: Depends on the template, usually a Google Map with a marker is shown. Therefore, the field needs to define two child fields `latitude` and `longitude` of type `decimal`:

```
<field name="coordinate" type="coordinate">
    <label>coordinate</label>
    <field name="latitude" type="decimal" format="#0.0000000" >
        <label id="latitude" />
    </field>
    <field name="longitude" type="decimal" format="#0.0000000" >
        <label id="longitude" />
    </field>
</field>
```

Table Mode: not supported

5.2.11. date

Target Java Type: `java.util.Date`

Form Mode: `<input type="text">` with the formatted string representation. The template usually adds a user friendly date picker widget.

Table Mode: formatted string representation (standard format: `yyyy-MM-dd HH:mm:ss`)

Supported format: a SimpleDateformat pattern, e.g. `format="yy/MM/dd"`

5.2.12. file

Target Java Type: org.appng.forms.FormUpload (form mode) or `String` (table mode)

Form Mode: `<input type="file">`

Table Mode: standard string representation and a file icon

In table mode, the Java target type is a `String`, representing the **name** of the file. The template can then display a file icon corresponding to the file extension.

5.2.13. `file-multiple`

Target Java Type: `java.util.Collection<org.appng.forms.FormUpload>`

Form Mode: `<input type="file" multiple="multiple">`

Table Mode: not supported

5.2.14. `image`

Target Java Type: `java.lang.String`, interpreted as the relative path to an image resource

Form Mode: `<img src="…">`

Table Mode: `<img src="…">`

5.2.15. `linkpanel`

Target Java Type: None, has a special meaning, see Linkpanels and Links for details.

5.2.16. List types

List types are mainly used in form mode to display checkboxes, radio buttons or selections. In order to do that, some special handling inside the `DataProvider<T>` is necessary. Generally speaking, for each field of a list type, a org.appng.xml.platform.Selection must be added to the `DataContainer` returned by a `DataProvider`. The **id** of the selection must match the **name** of the field, so the template can make the connection between them.

Let's make an example. In our bindclass, we have the following field:

```
private Collection<Integer> selectedIds = new ArrayList<Integer>();
```

This field is mapped in the datasource:

```
<field name="selectedIds" type="list:select">
    <label id="selectedIds" />
</field>
```

We want this field to be rendered as `<select multiple="multiple">` (multiple because a regular select can only have one value). What we have to do in the `DataProvider` now is:

```
DataContainer dataContainer = new DataContainer(fieldProcessor); ①
SelectionFactory selectionFactory = getSelectionFactory();       ②
Integer[] all =  new Integer[] { 1, 2, 3 };                      ③
DemoBean d = getDemoBean();                                      ④
Collection<Integer> selected = d.getSelectedIds();               ⑤
Selection selectedIds = selectionFactory
  .fromObjects("selectedIds", "selectedIds", allIds, selected);  ⑥
selectedIds.setType(SelectionType.SELECT_MULTIPLE);              ⑦
dataContainer.getSelections().add(selectedIds);                  ⑧
```

① create a `DataContainer` to be returned

② retrieve a `SelectionFactory`

③ define the values that can be selected

④ retrieve the bind-object

⑤ retrieve the selected ids

⑥ call one of the helper method of the `selectionFactory` to build a `Selection`. Those helper methods always take the id of the selection as the first argument. This id must match the field's name.

⑦ set the appropriate org.appng.xml.platform.SelectionType

⑧ add the `selection` to the `dataContainer`

As you see,the simplest way to create a selection is using one of the helper methods of the SelectionFactory. In most real world applications, you will build selections not from simple Java types, but from your domain model.

Therefore, it is a good idea to let your domain model classes implement org.appng.api.model.Identifiable or org.appng.api.model.Named. If done so, you can use one of the `fromIdentifiable(…)` or `fromNamed(…)` helper methods of org.appng.api.support.SelectionFactory. Also check the `fromEnum(…)` methods to be used with `java.lang.Enum`.

`list:checkbox`

Selection Type: `SelectionType.CHECKBOX`

Target Java Type: `java.util.Collection<[String|Integer]>`

Form Mode: multiple `<input type="checkbox">`

Table Mode: an HTML unordered list (`<ul>`) containing the collection values

`list:radio`

Selection Type: `SelectionType.RADIO`

Target Java Type: `String` or `Integer`

Form Mode: multiple `<input type="radio">`

Table Mode: standard string representation

`list:select`

Selection Type: `SelectionType.SELECT` / `SelectionType.SELECT_MULTIPLE`

Target Java Type: `[String|Integer]` / `java.util.Collection<[String|Integer]>`

Form Mode: `<select>` or `<select multiple="multiple">`

Table Mode: an HTML unordered list (`<ul>`) containing the single value/ the collection values

`list:text`

Selection Type: `SelectionType.TEXT`

Target Java Type: `java.util.Collection<String>`

Form Mode: multiple `<input type="text">`

Table Mode: an HTML unordered list (`<ul>`) containing the collection values

5.3. The `current` variable

In a datasource, the implicit variable `current` is always available. It the single-item-case, `current` refers to that single item. In case the datasource returns multiple items, you may consider `current` as the loop variable.

5.4. Linkpanels and Links

5.4.1. Adding inline links for each item

Like shown in the above example, it is easy to write a datasource that returns a collection of items. Now imagine for each of those items, you want to provide a link to edit or delete this item.

Therefore we need to add a `<field type="linkpanel">` to the datasource's `<meta-data>`. Because we can not define the linkpanel inside the field, we need to add it to the `<config>` element:

```
<datasource id="employees"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
        <title id="employees" />
        <params>
            <param name="selectedId" />
        </params>
        <meta-data bindClass="com.myapp.domain.Employee"
                result-selector="${current.id eq selectedId}">
            <field name="id" type="int">
                <label id="id" />
            </field>
            <field name="lastName" type="text">
                <label id="lastName" />
            </field>
            <field name="firstName" type="text">
                <label id="firstName" />
            </field>
            <field name="dateOfBirth" type="date" format="yyyy-MM-dd">
                <label id="dateOfBirth" />
            </field>
            <field name="actions" type="linkpanel"> ①
                <label id="actions" />
            </field>
        </meta-data>
        <linkpanel location="inline" id="actions"> ②
            <link target="/employees/update/${current.id}" mode="intern"> ③
                <label id="employee.update" /> ④
                <icon>edit</icon> ⑤
            </link>
            <link target="/employees&#63;delAction=delete&#38;delId=${current.id}"
mode="intern">⑥
                <label id="employee.delete" />
                <icon>delete</icon>
                <confirmation id="employee.delete.confirm"
                    params="${current.firstName},${current.lastName}" />⑦
            </link>
        </linkpanel>
    </config>
    <bean id="employees">
        <option name="mode" value="list" />
    </bean>
</datasource>
```

① We define a field of type `linkpanel`.

② Next, a linkpanel whose `id` **matches** the `name` of the linkpanel field, must be defined.

③ The link to edit an item uses `mode="intern"`, because it points to the page "employees". The page is responsible for including the action for editing the item. See Actions and Pages for more information on that topics.

④ A link has a label, that is used for a mouseover tooltip.

⑤ A link can use from a set of predefined icons that each appNG Template must provide. See List of icons for a list of predefined icons.

⑥ The link to delete the item uses some GET-parameters. Please note that for the question mark (?) and the ampersand (&), the corresponding Numeric character references `&#63;` and `&#38;` must be used in order to not break the XML document during XSLT transformation.

⑦ Because we want to avoid accidentally deleting an item, we add a parametrized `<confirmation>`. The entry in the application's dictionary would look like this: `employee.delete.confirm=Delete {0} {1}?`

5.4.2. Adding links for the whole datasource

Finally, a link to create a new item should be provided by the datasource. This is done by adding a linkpanel with location `top`, `bottom` or `both`. **No field of type linkpanel must be added to the meta-data!**

```
<datasource id="employees">
    <config>
    ...
    <meta-data>
    ...
    </meta-data>
        <linkpanel location="both" id="other">
            <link target="/employees/create" mode="intern">
                <label id="employee.create" />
                <icon>new</icon>
            </link>
        </linkpanel>
    </config>
    ...
</datasource>
```

The link generated this way is shown above/under/above and under the resulting table, depending on the chosen `location` (one of `top`, `bottom`, `both`).

5.4.3. Links to webservices

If you want to provide a link to a webservice, use the following approach:

```
<link mode="webservice" target="downloadService?id=${current.downloadId}"> ①
  <label id="download" />
  <icon>download</icon>
</link>
```

① Use the `webservice`-mode for the link. The target starts with the name of the bean implementing the webservice, followed by the parameters required by that service.

More about webservices can be found in this chapter.

5.5. Field references

As an alternative to the `current`-variable, there's the **field reference syntax**. As a prefix, it uses the number/hash-sign (`#`) instead of the dollar sign (`$`). With this syntax, you can reference field-values **by their name**.

Field references can be used

- inside inline-links
  - in the `target`-attribute
  - as a `<label>`-parameter
  - as a `<confirmation>`-parameter
- as a `<title>`-parameter of an `<action>`, referencing a single-item datasource

This said, the datasource employees could also define it's links like shown below:

```
<link target="/employees/update/#{id}" mode="intern">
    <label id="employee.update" params="#{firstName}, #{lastName}"/>
    <icon>edit</icon>
</link>
<link target="/employees&#63;delAction=delete&#38;delId=#{id}" mode="intern">
    <label id="employee.delete" params="#{firstName}, #{lastName}"/>
    <icon>delete</icon>
    <confirmation id="employee.delete.confirm" params="#{firstName}, #{lastName}" />
</link>
```

Then, the resourcebundle would look like

```
employee.update = Edit {0} {1}
employee.delete = Delete {0} {1}
employee.delete.confirm = Really delete {0} {1}?
```

5.6. Paging, Sorting and Filtering

5.6.1. Paging and Sorting

Sorting is a first class citizen of the appNG application platform. If you use the org.springframework.data.domain.Page abstraction of Spring Data JPA together with repository methods that take a org.springframework.data.domain.Pageable object as an argument, it is easy to provide those features.

For paging, nothing needs to be changed in the source XML of the datasources. Just call fieldProcessor.getPageable() and use the returned `Pageable` as a method argument for a repository search method. Within the page, you can use the `pageSize`-attribute of the `<datasource>` to change the default page size of 25 for that datasource.

If you can not/do not want to provide paging and sorting by implementing it inside your business logic, you can use DataContainer.setPage(Collection<?> items, Pageable pageable) and let the framework handle it.
You can even disable paging and sorting by using DataContainer.setItems(Collection<?> items)

For (multi-column) sorting, just add the `<sort />`-element to the `<field>` you want to sort by, **and that's it!**. For each field that is sortable, the sort icons will be displayed in the table header for that field.

The `<sort/>` element offers the following attributes:

- `prio` (`integer`)
  The priority for the default-sort, less means higher. Start with 0 and increase by 1.
- `order` ([`ASC`|`DESC`])
- `order` ([`ASC`|`DESC`])
  The direction for the default-sort
- `name` (`string`)
  The name of the property to be used in a JPA search query. Only necessary if different from the field's name.
- `ignore-case` (`boolean`)
  Whether or not the search should be case insensitive or not.

If you want to define a default sort order for a datasource, just use the `prio` and order `attributes` . In the following example, we define a default sorting by last name and first name, both descending and ignoring case:

```
<field name="lastName" type="text">
     <sort order="DESC" prio="0" ignore-case="true" />
<field>
<field name="firstName" type="text">
     <sort order="DESC" prio="1" ignore-case="true" />
<field>
```

5.6.2. Filtering

If your application is dealing with large sets of data, the ability to filter this data is a common requirement. That's why appNG offers the concept of filters. A filter is a `SelectionGroup` that is added to a `DataContainer`. To that `SelectionGroup`, you add your filter criteria in form of a `Selection`, obtained from a `SelectionFactory`. See the section about building selections for details.

Filtering is done by rendering a HTML `<form>` containing the filter criteria. That form submits its data with the HTTP `GET`-method, which leads to bookmarkable filter-criteria. Although there are many different ways how to implement the filter functionalities in detail, there has evolved some kind of best practices for this.

If your datasource only requires a few filter criteria, it is feasible to manually read them from the request as shown below:

```
import org.apache.commons.lang3.StringUtils;
...
public DataContainer getData(...) {
  DataContainer dataContainer = new DataContainer(); ①
  SelectionGroup group = new SelectionGroup();
  dataContainer.getSelectionGroups().add(group);

  String filterByName = request.getParameter("nameFilter"); ②
  Selection nameFilter =
    selectionFactory.fromObjects("nameFilter", "label.nameFilter",
      new String[] { StringUtils.trimToEmpty(filterByName) });
  nameFilter.setType(SelectionType.TEXT);
  group.getSelections().add(nameFilter);③

  Page<Employee> employees =
    service.searchEmployees(name, fieldProcessor.getPageable());
  dataContainer.setPage(employees);
  return dataContainer; ④
}
```

① create a `DataContainer` and add a `SelectionGroup` (read: filter)

② manually read the value of the filter criteria from the request

③ use a `SelectionFactory` to create a `Selection`, respecting the current value taken from the request

④ retrieve the data and return it

Advanced Filtering Techniques

In most cases where filtering is required, more than one filter criteria is available for the user. Since we do not want to manually read the string values from the request and if necessary convert them to the right type, another approach can be taken.

1. Define and implement a `DataProvider` that wraps your filter criteria

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS) ①
public class Filter implements DataProvider {

  private String nameFilter; ②
  private Integer idFilter;

  public DataContainer getData(Site site, Application application,
      Environment environment, Options options,
      Request request, FieldProcessor fp) {
    try {
      request.fillBindObject(this, fp, request, site.getSiteClassLoader()); ③
      DataContainer dataContainer = new DataContainer(fp);
      dataContainer.setItem(this);
      return dataContainer; ④
    } catch (BusinessException e) {
      throw new ApplicationException("error filling filter", e);
    }
  }

  // getters and setters here
}
```

① The filter needs to be **request**-scoped

② define the filter criteria

③ do a `request.fillBindObject(…)` with the `Filter` itself

④ return a `DataContainer`

Note the `proxyMode` for the filter is set to `ScopedProxyMode.TARGET_CLASS`. This is necessary if you want to inject the filter into a bean with a longer-lived scope. Because a "regular" `DataProvider` uses singleton scope, we need the make the filter a scoped proxy. Check out the Spring Reference Documentation for more details about scoped proxies.

The corresponding source XML definition:

```
<datasource id="filter"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
  <config>
    <meta-data bindClass="com.myapp.business.Filter">
      <field name="nameFilter" type="text">
        <label id=nameFilter" />
      </field>
      <field name="idFilter" type="int">
        <label id="idFilter" />
      </field>
    </meta-data>
  </config>
  <bean id="filter" />
</datasource>
```

2. Include this datasource in your page
   The important thing here is to include the filter datasource **before** the datasource you want to filter. Because there is no need to display this datasource, we mark the `<section>` as `hidden`.

```
<section hidden="true">
  <element>
    <datasource id="filter" />
  </element>
</section>
```

3. In the datasource to be filtered, inject the filter and build your selections from it

```
  private SelectionFactory selectionFactory;

  private Filter filter;

  @Autowired
  public DemoBeans(SelectionFactory selectionFactory, Filter filter) {
    this.selectionFactory = selectionFactory;
    this.filter = filter;
  }

  public DataContainer getData(...) {
    DataContainer dataContainer = new DataContainer(fp);
    SelectionGroup group = new SelectionGroup();
    // build selections with a SelectionFactory and add them to the group
    // group.getSelections().add(selection);
    dataContainer.getSelectionGroups().add(group);
  }
```

5.7. Datasource inheritance

A datasource can inherit from another datasource. This can be defined by a special datasource id notation where the *id* of the inherited datasource is appended with two double colon after the datasource id `<datasource id="child::parent">`.
If a datasource id contains the inheritance separator `"::"`, appNG clones the parent datasource definition and uses the first part of the id as id for the final datasource. Other elements of the datasource are added or overwritten:

- **title**
  If the child defines another title than the parent datasource, the title gets overwritten. Otherwise the title of the parent datasource is used.
- **parameter**
  The list of parameters are copied from the inherited datasource. Parameters, defined in the child datasource, are added to the existing list of parameters. If the child has a parameter with the same name as a parameter from parent datasource, the parameter gets overwritten by the child datasource definition.
- **bind class**
  The bind class definition is mandatory anyhow. The child can define another bind class than the parent datasource. This is possible, if the bind class provides the same attributes referenced in the datasource fields, conditions and links as the original bind class of the parent datasource.
- **fields**
  The list of fields are copied from the parent datasource. New fields will be prepended to the list of existing fields. If the parent datasource has a linkpanel-field, the new fields will be prepended to the linkpanel-field. Fields with the same name will overwrite existing fields which have been copied from the parent datasource.
- **bean id**
  The complete bean definition is copied from the parent datasource. If the child defines another bean id than the parent datasource, the bean id gets overwritten.
- **bean options**
  The list of bean options are copied from the parent datasource. Bean options in the child datasource are added to the existing list. If the child datasource has an bean option with the same option name, the inherited option gets overwritten. Bean options are also inherited if the bean id is different.
- **linkpanels**
  The list of linkpanels is copied from the parent datasource. If the child provides additional linkpanels, the panels are added to the list from the parent datasource.

  If the child provides a link panel with the same id as an existing link panel, the linkpanel from parent datasource and all its links gets overwritten by the linkpanel definition of the child datasource.
- **config permissions**
  Permissions are copied from the parent datasource. If the child datasource defines it's own permissions, those from the parent datasource get overwritten.

- **config labels**
  The list of labels are copied from the parent datasource. New labels provided by the child are added. If the child datasource provides a label with the same id that an inherited label, the inherited label gets overwritten overwritten.

A datasource definition which has only an id-attribute and no other elements is not valid. Though it is necessary to add at least the mandatory elements. The simplest possible example would be:

```
<datasource id="employees2::employees"  ①
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>②
      <meta-data bindClass="com.myapp.domain.Employee" /> ③
    </config>
    <bean /> ④
</datasource>
```

① datasource `employees2` inherits datasource `employees`

② a datasource must have a config node

③ a config node must have a meta-data node with bindClass attribute. It uses the same bind class as the parent datasource

④ a datasource must have a bean node

The datasource can be referenced with id `employees2` in pages and actions. It is an exact clone of the datasource `employees` just with another id. This new datasource will not provide any additional elements or features compared to the parent datasource.

Even this simple clone is useful. It makes it possible to place the same datasource with different parameters on the same page because of the different ids.

An example for a more complex inheritance would look like this:

```
<datasource id="teamleads::employees"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
        <title id="teamleads" /> ①
        <meta-data bindClass="com.myapp.domain.TeamLead" > ②
            <field name="teamSize" type="int"> ③
                <label id="teamSize"/>
            </field>
        </meta-data>
        <linkpanel location="inline" id="actions"> ④
            <link target="/teamlead/showteam/${current.id}" mode="intern">
                <label id="teamlead.team" />
                <icon>user</icon>
            </link>
        </linkpanel>
    </config>
    <bean>
        <option name="kind" value="teamlead" /> ⑤
    </bean>
</datasource>
```

① The datasource gets a new title.

② The bind class is now `TeamLead` which has to provide all referenced attributes from the parent datasource. It also has to provide all attributes referenced in this child datasource

③ The datasource gets an additional field showing the size of the team

④ The linkpanel with `id="actions"` of the datasource employee gets overwritten. It will now only provide one link to show the teamlead's team members. All inherited links has been removed with the inherited linkpanel. If the other links are required, they must be duplicated in the child definition.

⑤ This adds an option to the inherited bean option list.

After appNG processed the inheritance, the final datasource will have the following structure:

```
<datasource id="teamleads"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
        <title id="teamleads" />
        <params>
            <param name="selectedId" />
        </params>
        <meta-data bindClass="com.myapp.domain.TeamLeads"
                result-selector="${current.id eq selectedId}">
            <field name="id" type="int">
                <label id="id" />
            </field>
            <field name="lastName" type="text">
                <label id="lastName" />
            </field>
            <field name="firstName" type="text">
                <label id="firstName" />
            </field>
            <field name="dateOfBirth" type="date" format="yyyy-MM-dd">
                <label id="dateOfBirth" />
            </field>
            <field name="teamSize" type="int">
                <label id="teamSize"/>
             </field>
            <field name="actions" type="linkpanel">
                <label id="actions" />
            </field>
        </meta-data>
        <linkpanel location="inline" id="actions">
            <link target="/teamlead/showteam/${current.id}" mode="intern">
                <label id="teamlead.team" />
                <icon>user</icon>
            </link>
        </linkpanel>
    </config>
    <bean id="employees">
        <option name="mode" value="list" />
        <option name="kind" value="teamlead" />
    </bean>
</datasource>
```

Because the bean id is not overwritten by the child datasource, the called bean for this datasource is still the bean `employees`. An example enhancement for the class `Employees` using the additional option `kind` would be:

```
// (...)
@Component
@Scope("request")
public class Employees implements DataProvider {
    //(...)

    public DataContainer getData(Site site, Application application, Environment
environment, Options options, Request request, FieldProcessor fp) {

        DataContainer dataContainer = new DataContainer(fp);
        String mode = options.getOptionValue("mode", "value");
        String kind = options.getOptionValue("kind", "value"); ①
        if ("list".equals(mode)) {
            if("teamlead".equals(kind){
                Page<TeamLead> teamLeads = service.findTeamLeads(fp.getPageable()); ②
                dataContainer.setPage(teamLeads);
            } else {
                Page<Employee> employees = service.findEmployees(fp.getPageable());
                dataContainer.setPage(employees);
            }
        } else {
            // do something else
        }
        return dataContainer;
    }
    // (...)
```

① Retrieve the option `kind`. If the bean has been called for datasource `employees`, the variable will be null. If it has been called for datasource `teamleads` the value will be `"teamlead"`.

② Get the team leads instead of all employees and put them in the data container.

The `FieldProcessor` provides the datasource name as `String` with method `fp.getReference()`. This could also be used instead adding the option `kind` in the example above.

It would also be possible to define a new DataProvider bean (TeamLeads) to provide data for this datasource instead enhancing the existing bean `Employees`.

5.7.1. Multiple inheritance

Multiple inheritance is not supported. A datasource can have only one parent. But it is possible to build an inheritance hierarchy:

```
<datasource id="employees2::employees" ①
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
      <meta-data bindClass="com.myapp.domain.Employee" />
    </config>
    <bean />
</datasource>

<datasource id="employees3::employees2 ②
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <config>
      <meta-data bindClass="com.myapp.domain.Employee" />
    </config>
    <bean />
</datasource>
```

① employees2 inherits employees

② employees3 inherits employees2 (which inherited employees)

AppNG will first process the inheritance of `employees` into `employees2` with all rules defined above. After that it will process the inheritance of `employees2` into `employees3` with the same rules no matter that this datasource already inherited another datasource.

5.8. Datasources as a service

It is possible to retrieve a single datasource with a special service URL. The schema for such an URL is

```
http(s)://<host>[:<port>]/service/<site-name>/<application-
name>/datasource/<format>/<datasource-id>
```

The supported formats are **xml** and **json**.

Examples:

- http://localhost:8080/service/manager/myapp/datasource/xml/mydatasource
- http://localhost:8080/service/manager/myapp/datasource/json/mydatasource?param1=foo&param2=bar

As you can see in the second example, you can pass parameters defined by the datasource as GET-parameters.

Datasources that should be available through a service URL **must** be secured by a permission. If no permission is present, the access will be denied. Anonymous permissions might be used.

5.8.1. Defining paging and sorting

When used as a service, paging and sorting capabilities can also be used by a specially named GET-Parameter. As an example, let's take the datasource `employees` as defined here. To sort this datasource by *lastname* ascending and by *firstname* descending and to retrieve the *second* page with a page size of *25*, the URL would be:

http://localhost:8080/service/manager/myapp/datasource/json/employees?sortEmployees=page:1;pageSize:25;firstName:asc;lastName:asc

Let's further analyse the query-part of the URL. It contains a **single GET-Parameter** that defines the above mentioned criteria for paging and sorting. The syntax is `<criterion>:<value>`, where multiple criteria are separated with a semicolon (`;`).

```
sortEmployees= ①
page:1;        ②
pageSize:25;   ③
firstName:asc; ④
lastName:desc  ⑤
```

① The name of the parameter is `sort` plus the capitalized id of the datasource.

② Since appNG uses *zero-indexed* pages, the number of the second page is 1

③ Define the size of the page.

④ Sort by firstname, ascending.

⑤ Sort by lastName, descending.

Since appNG supports multi-field-sorting, the order of the fields within the sort-parameter is essential.

6. Actions

While a datasource represents the structure/state of the application's data, actions are there to manipulate the state of that data. In most cases, an action will be rendered as an HTML `<form>` with various `<input>`-fields.

Let's see how a typical action looks like:

```
<event id="employeeEvent"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd"> ①
    <config></config>
    <action id="update"> ②
        <config>
            <title id="employee.update" /> ③
            <params> ④
                <param name="id" />
                <param name="form_action" />
            </params>
        </config>
        <condition expression="${'update' eq form_action and not empty id}" /> ⑤
        <datasource id="employee"> ⑥
            <params>
                <param name="id">${id}</param>
            </params>
        </datasource>
        <bean id="employees"> ⑦
            <option name="mode" value="update" id="${id}" />
        </bean>
    </action>
</event>
```

① Actions are always grouped by an `<event>`. That makes it easy to collect all actions that perform on the same domain object within on event. The id of the event **must** be globally unique.

② Within the event, each action **must** have a unique id.

③ An action **must** define a title that is used as a headline when rendered as HTML.

④ An action **can** be parametrized using a `<params>` element.

⑤ An action **can** define an execute condition. Only if the condition matches, the action will be executed.

⑥ An action **can** make use of a datasource and pass its parameters to that datasource.

⑦ Just like for datasources, the name of the Spring bean implementing the action **must** be given. Also, any number of options can be passed to that implementation.

As you see, we use the datasource employee defined before. Thus the type of the data received by the action is an `Employee`.

Next, the implementation:

```
import org.appng.api.ActionProvider;
import org.appng.api.Environment;
import org.appng.api.FieldProcessor;
import org.appng.api.Options;
import org.appng.api.Request;
import org.appng.api.model.Application;
import org.appng.api.model.Site;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import com.myapp.domain.Employee;
import com.myapp.service.EmployeeService;

@Component
@Scope("request") ①
public class Employees implements ActionProvider<Employee> { ②

    private EmployeeService service;

    @Autowired
    public Employees(EmployeeService service) {
        this.service = service;
    }

    ③
    public void perform(Site site, Application application,
            Environment environment, Options options, Request request,
            Employee formBean, FieldProcessor fieldProcessor) {
        ④
        String mode = options.getOptionValue("mode", "value");
        Integer id = request.convert(options.getOptionValue("mode", "id"), Integer
.class);
        if ("update".equals(mode)) {
            ⑤
            service.updateEmployee(formBean, request, fieldProcessor);
        } else {
            ...
        }
    }
}
```

① Spring annotations are used to define the bean.

② The interface to be implemented is org.appng.api.ActionProvider<T>. Because the datasource uses the bindclass `Employee`, we use this as type parameter.

③ The single method to implement is `perform(…)`, which takes a parameter `formBean` of the required type `Employee`.

④ We retrieve the mode and the id from the options.

⑤ We call the service to update the employee.

As you can see, there is no need for manually binding request parameters to the `formBean`, appNG is doing that job for you. Now, you might ask how the implementation of `service.updateEmployee(…)` might look like and why it needs the `request` and `fieldProcessor` arguments. Good catch!

In web applications, we always face the problem that the data received from the user has to be merged with the real, probably persistent data. This problem even gets worse as the user most times is only allowed to edit certain properties of the data. So what to do? Merge each property manually? This is not the appNG way.

But this is:

```
public void updateEmployee(Employee formBean, Request request, FieldProcessor fp) {
    Employee current = getEmployee(formBean.getId()); ①
    request.setPropertyValues(formBean, current, fp.getMetaData()); ②
    String message =
        request.getMessage("employee.updated", e.getFirstName(), e.getLastName()); ③
    fp.addOkMessage(message);
}
```

① We retrieve the current data.

② For each non-readonly (aka writable) field of the `FieldProcessor`, the field value is written from the `formBean` to the current data.

③ Retrieve a parametrized message from the resourcebundle and add it to the `FieldProcessor`.

How does this work? Well, the `FieldProcessor` contains the org.appng.xml.platform.MetaData with all the org.appng.xml.platform.FieldDefinitions defined in the datasource employee. Because of this, the `request` can easily write those fields to the target object, including possible necessarry type conversions.

6.1. Validation

Validation is a first class citizen in the appNG application framework. Therefore, the Bean Validation API 2.0 as specified in **JSR-349** is used. Just add some validation annotations to the POJO bindclass that you datasource uses. As long as the `formBean` of an `ActionProvider` is not valid, the action will not perform and the error messages will be shown.

Example:

```
@NotNull(message = ValidationMessages.VALIDATION_NOT_NULL) ①
@Size(min = 3, message = "{firstname.toShort}") ②
public String getFirstName() {
    return firstName;
}
...
@FileUpload(fileTypes = "jpg,png", maxSize = 10, unit = Unit.MB) ③
public org.appng.forms.FormUpload getUpload() {
    return upload;
}

@Valid
public Salary getSalary() { ④
    return salary;
}
```

① A predefined message from org.appng.api.ValidationMessages can be used

② Alternatively, a custom validation message from the application's resourcebundle can be used. Don't forget the curly braces!

③ A org.appng.forms.FormUpload can be validated using `@FileUpload`.

④ Nested properties can be validated using `@Valid`.

You can use any Constraint in your application, may it be the standard constraints from the `javax.validation.constraints`-package, those from Hibernate Validator (shipped with appNG), or any custom and/or compound constraints.

6.1.1. Client side validation

An appNG template ships with support for client-side validation. Because it can not know all possible constraints you are using in your action's bindclass, only these standard constraints are eligile for client-site validation:

- javax.validation.constraints.Digits
- javax.validation.constraints.Future
- javax.validation.constraints.Max
- javax.validation.constraints.Min
- javax.validation.constraints.NotNull
- javax.validation.constraints.Past
- javax.validation.constraints.Pattern
- javax.validation.constraints.Size
- org.appng.api.FileUpload
- org.appng.api.NotBlank

If your action's bindclass uses some additional constraints, it should be considered to set `clientValidation="false"` for your `<action>`. Otherwise, the client-validation would pass, whereas the server-side validation fails. This could be confusing to the user.

6.1.2. Programmatic validation

By implementing org.appng.api.FormValidator in your `ActionProvider`/ `DataProvider` **or** in your bind-object, validation can be done programmatically.

Example:

In this example, a credit card number is validated inside the bind-object `BankAccount` using Apache Commons Validator's `CreditCardValidator`.

```
import org.apache.commons.validator.routines.CreditCardValidator;
import org.appng.api.Environment;
import org.appng.api.FieldProcessor;
import org.appng.api.FormValidator;
import org.appng.api.Options;
import org.appng.api.Request;
import org.appng.api.model.Application;
import org.appng.api.model.Site;
import org.appng.xml.platform.FieldDef;

public class BankAccount implements FormValidator { ①

    private String creditCardNumber;

    ②
    public void validate(Site site, Application application,
            Environment environment, Options options,
            Request request, FieldProcessor fieldProcessor) {
        ③
        if (!new CreditCardValidator().isValid(creditCardNumber)) {
            FieldDef creditCardNumber = fieldProcessor.getField("creditCardNumber");
④
            fieldProcessor.addErrorMessage(creditCardNumber, "Invalid credit card
number!"); ⑤
        }
    }

    // getters and setters here

}
```

① Implement `org.appng.api.FormValidator`.

② The single method to be implemented is `validate(…)`.

③ Check if the credit card number is valid.

④ If not, get the `FieldDef` for `creditCardNumber` from the `FieldProcessor` by it's `binding`.

⑤ Add an error-message for the field.

You can also use the `FieldProcessor` inside your `ActionProvider`/ `DataProvider` to add validation error messages to a field or to the whole action/datasource respectively. In case of an action, make sure not to process any further and just `return` from the action.

Example:

```
public SaveBankAccount implements ActionProvider<BankAccount> {

    public void perform(Site site, Application application,
            Environment environment, Options options, Request request,
            BankAccount formBean, FieldProcessor fieldProcessor) {
        ①
        if (!new CreditCardValidator().isValid(formBean.getCreditCardNumber())) {
            FieldDef creditCardNumber = fieldProcessor.getField("creditCardNumber");
②
            fieldProcessor.addErrorMessage(creditCardNumber, "Invalid credit card
number!");
            fieldProcessor.addErrorMessage("Your input contains errors!"); ③
            return; ④
        }
        // continue
    }
}
```

① Check if the credit card number is valid.

② If not, add the error-message to the `creditCardNumber` field.

③ Add a global error message.

④ Stop processing and return.

6.1.3. Using validation groups

The Bean Validation API supports the concept of validation groups, as can bee seen in section 5.1.3 that specification.

AppNG does support validation groups by adding those to the `<meta-data>` of an application's datasource. Usually, an interface defined inside the class to be validated is used as a validation group.

Example:

```
<meta-data bindclass="com.myapp.domain.Employee">
    <validation>①
        <group class="javax.validation.groups.Default" /> ②
        <group class="com.myapp.domain.Employee$AddressFields"
            condition="${current.active}" /> ③
    </validation>
    <!-- list of <field> elements follows here -->
</meta-data>
```

① Add a `<validation>`-element to the `<meta-data>` to define groups.

② Adds the `Default` group.

③ Adds a custom validation group. If this is an inner interface, the dollar sign (`$`) needs to be used to specify the fully qualified class name for the group. The `condition` tells the framework only to use this group if the (boolean) property `active` of current `Employee` is `true`.

6.2. Actions as a service

It is possible to retrieve and perform a single action with a special service URL. The schema for such an URL is

`http(s)://<host>[:<port>]/service/<site-name>/<application-name>/action/<format>/<event-id>/<action-id>`

The supported formats are **xml** and **json**.

Examples:

- http://localhost:8080/service/manager/myapp/action/json/myEvent/myAction
- http://localhost:8080/service/manager/myapp/action/xml/myEvent/myAction

In order to execute the action, the client of the webservice must send an HTTP-POST request containing all the mandatory fields plus the action parameters that are required to satisfy the execute-`<condition>` of the action.

Actions that should be available through a service URL **must** be secured by a permission. If no permission is present, the access will be denied. Anonymous permissions might be used.

7. Pages

7.1. `<page>` and `<pages>`

A `<page>` is a single view presented to the user. On a page you can compose several Actions and Datasources within any number of sections and section elements.

An example of a page is shown below:

```
<page id="employees"
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd"> ①
    <config>
        <title id="page.employees" />
        <url-schema> ②
            <url-params>
                <url-param name="action" />
                <url-param name="id" />
            </url-params>
            <get-params>
                <get-param name="delAction" />
                <get-param name="delId" />
            </get-params>
            <post-params> ③
                <post-param name="form_action" />
            </post-params>
        </url-schema>
    </config>
    <structure>
        <section>
            <element>
                <action id="create" eventId="employeeEvent"> ④
                    <condition expression="${action eq 'create'}" />
                    <params>
                        <param name="form_action">${form_action}</param>
                    </params>
                </action>
            </element>
            <element>
                <action id="update" eventId="employeeEvent"> ⑤
                    <condition expression="${action eq 'update' and not empty id}" />
                    <params>
                        <param name="id">${id}</param>
                        <param name="form_action">${form_action}</param>
                    </params>
                </action>
            </element>
            <element>
                <datasource id="employees"> ⑥
                    <params>
                        <param name="selectedId">${id}</param>
                    </params>
                </datasource>
            </element>
        </section>
        <section hidden="true"> ⑦
            <element>
```

```
                <action id="delete" eventId="employeeEvent" onSuccess="/employees"> ⑧
                    <params>
                        <param name="action">${delAction}</param>
                        <param name="id">${delId}</param>
                    </params>
                </action>
            </element>
        </section>
    </structure>
</page>
```

① Each page **must** have a unique id.

② A page **must** define die URL schema.

③ Since a page can not 'know' every possible POST parameter only the built-in parameter `form_action` needs to be set here.

④ The `create` action is included, but only if `${action eq 'create'}`.

⑤ The `update` action is included, but only if `${action eq 'update' and not empty id}`.

⑥ Always include the 'employees' datasource, passing the `id` as a parameter.

⑦ Include a hidden section with the 'delete'-action, passing the get-parameters `delId` and `delAction`.

⑧ After a successful delete action, the user gets redirected to the page `employees`. This is done to cleanup the URL from the GET-parameters used for deletion.

7.2. <url-schema>

As shown in the example above, the current URL in conjunction with the URL-schema of the page is responsible for what action and datasources are being shown on that page.

As you already might know, an appNG application URL has the following form:

`http(s)://<host>[:<port>]/<manager-prefix>/<site>/<application>/<page>/<url-param-1-to-n>?<get-params>`

Example:

`http://localhost:8080/manager/appng/myapp/employees/update/7`

So with this URL, the runtime-representation of the URL-schema is:

```
<url-schema>
    <url-params>
        <url-param name="action">update</url-param>
        <url-param name="itemid">7</url-param>
    </url-params>
    <get-params>
        <get-param name="foo" />
    </get-params>
    <post-params>
        <post-param name="form_action" />
    </post-params>
</url-schema>
```

When rendering the page, appNG first checks the include conditions for every action and datasource. In the next step, every action whose execute-condition matches is beeing executed. This includes the execution of the datasources referenced by those actions. Finally, the datasources included on the page are processed.

7.3. <applicationRootConfig> and <navigation>

There is a single place where some general configuration for all pages must be done. There must be one source XML file containing an <applicationRootConfig>. The only exception for that rule is, if the application does not offer a GUI at all, but only offers some other services. See Implementing services for an overview about what services this could be.

Anyhow, if you build an application with a GUI, you need to provide an <applicationRootConfig>. A typical exemplar of that looks like this:

```
<applicationRootConfig
    xmlns="http://www.appng.org/schema/platform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.appng.org/schema/platform
        http://www.appng.org/schema/platform/appng-platform.xsd">
    <name>My Cool App</name>①
    <config>
        <session>
            <session-params>②
                <session-param name="param1" />
                <session-param name="param2" />
            </session-params>
        </session>
    </config>
    <navigation location="top" id="topnav">③
        <link mode="intern" target="/employees">
            <label id="employees" />
        </link>
        <link mode="intern" target="/settings">
            <label id="settings" />
        </link>
    </navigation>
</applicationRootConfig>
```

① provide a name, just for informational purposes

② define which post-/get-/url-parameters should be persisted in the user's HTTP session

③ provide the top level navigation for the application, linking to certain pages

Note that each page page-parameter -**may it be a GET-, POST- or URL-parameter**- that is defined as a session-parameter, will be saved into the session and does not change until a different value is assigned or the session expires. If a page defines a parameter that it not contained in the current request, but there is a session parameter with that name, the value from the session will be taken.

A common use-case for session parameters is using them for filter criteria of a datasource. See here for more about filtering.

With great power comes great responsibility!
Using session parameters is a powerful feature. Decide carefully which parameters should get stored in the session.

8. Expressions

AppNG uses the JSP Expression language syntax to provide support for dynamic expressions. See here for a list of operators.

Expressions can be used to

- pass page-parameters to an `<action>` / `<datasource>`

- pass `<action>`-parameters to a `<datasource>`
- pass `<action>`/`<datasource>`-parameters to the attributes of `<option>` elements
- control the `readonly`-, `hidden`- and `format`-attribute of a `<field>`
- write the `expression` for a `<condition>` of a `<field>`,`<action>`,`<datasource>`,`<link>`
- set the `passive`- and `folded`-attributes of a section `<element>`
- set the `hidden`-attribute of a `<section>`
- set the `async`-attribute of an `<action>`
- set the `default` and `active`-attribute of a `<link>`
- write a `condition` for an `<icon>`
- pass a parameter to the `params`-attribute of a `<label>`, `<title>`, `<description>`, `<confirmation>`

9. Permissions

As described here, an application can define the available permissions it ships with in `application-home/application.xml`. Those permissions can be referenced at many places inside the source XML documents.

Those places are:

- the `<config>`-element of a(n)
  - `<page>`
  - `<action>`
  - `<datasource>`
  - `<event>`
- a `<link>`/`<linkpanel>`
- an `<output-format>` and `<output-type>`
- a `<field>`

When referencing a permission at one of those places, the `mode`-attribute needs to be set. There are two different modes:

- `set`
  Determines if the current subject has the required permission, and **applies** it to the defining parent element. This means, if the permission is not present, the parent element will **not be contained** in the resulting target XML.

  This is the mode to be used in 99.99% percent of cases.

- `read`
  Reads, **but does not apply**, the required permission from the current subject. This means, even if the permission is missing, the parent element will **be contained** in the target XML.
  This mode can be used if some Custom XSL stylesheets should get applied, depending on the permissions of the subject.

9.1. Anonymous permissions

There is a special kind of permissions to allow anonymous access, using the prefix `anonymous`. Every user, regardless whether he's logged on or not, owns these permissions. Permissions that use the prefix `anonymous` **must not** be defined in `application.xml`.

Anonymous permissions are especially useful when providing access to actions and datasources as a service. See Actions as a service and Datasources as a service for more details on these topics.

9.2. Field permissions

For a `<field>` using permissions, it can be differentiated between reading and writing the field. Here, the `<permissions>`-element allows an additional `mode`-attribute, with the possible values `read` and `write`.

The following example shows how to use this mode:

```
<field name="foobar" type="text">
    <label id="foobar" />
    <permissions mode="read">
        <permission ref="foobar.read" mode="set" />
    </permissions>
    <permissions mode="write">
        <permission ref="foobar.write" mode="set" />
    </permissions>
</field>
```

9.3. Programmatically checking permissions

In cases where the referencing permissions in the static XML sources is not sufficient, it is also possible to check them programmatically.

Therefore, you can retrieve a org.appng.api.PermissionProcessor from a org.appng.api.Request. Then, use the `hasPermission(String reference)`-method to check for a certain permission:

```
if (request.getPermissionProcessor().hasPermission("doSomething")) {
  // do something
}
```

10. JSP Tags

AppNG provides a number of tags to be used in JavaServer Pages. Those can be used if an appNG site is used to serve JSP based content. Usually, this content originates from a content management sytem.

In order to use the appNG JSP tags, a `@taglib` directive must be added to the page:

```
<%@taglib uri="http://appng.org/tags" prefix="appNG"%>
```

Within JSP files, you can make use of the available *implicit objects* like `${param}`, `${requestScope}` `${sessionScope}`. For further details see section 15.12.2 of the The Java EE Tutorial.

10.1. <appNG:taglet>

This is probably the most common way of embedding dynamic contents provided by an appNG application into a JSP page. First, implement one of the following interfaces in your application and declare the implementing class as a Spring bean:

- org.appng.api.Taglet
- org.appng.api.GlobalTaglet
- org.appng.api.XMLTaglet
- org.appng.api.GlobalXMLTaglet

Example:

```
<appNG:taglet application="myapp" method="myTaglet">①
   <appNG:param name="foo">bar</appNG:param>②
   <appNG:param name="jin">fizzbar</appNG:param>
</appNG:taglet>
```

① Calls the taglet, `method` refers to the Spring bean name.

② Adds some parameters that are being passed as a `java.util.Map<String,String>` to the `processTaglet()`-method of the respective implementation.

Note that the content-type delivered by a taglet is variable. It can be XML, JSON, HTML or even plain text. See the Javadoc of the interfaces mentioned above for more details about the different types of taglets.

10.2. Form tags

With the form tags it it possible to build HTML forms, validate them on the server side and pass the user input to a org.appng.api.FormProcessProvider.

Example:

```
<appNG:form>①
   <appNG:formData mode="not_submitted">②
     <form action="" method="post" enctype="multipart/form-data" >③
        ④
        <appNG:formElement rule="email"
                mandatory="true"
                mandatoryMessage="E-mail is mandatory!"
                errorClass="error"
                errorMessage="Not a valid e-mail!"
                errorElementId="emailError">
            <input type="text" name="email" />⑤
            <div id="emailError">
        </appNG:formElement>
        <input type="submit" />
     </form>
   </appNG:formData>
   <appNG:formConfirmation application="appng-webutils" method="debugProvider" mode=
"submitted">⑥
     <appNG:param name="foo">bar</appNG:param>⑦
     <appNG:param name="jin">fizz</appNG:param>
     <p>Thank you for your message!<p>
   </appNG:formConfirmation>
 </appNG:form>
```

① Start with `<appNG:form>`.

② Next, define the content of the form using `<appNG:formData>`. The `mode` controls whether the form is always shown (`always`) or only as long as it hasn't been submitted (`not_submitted`).

③ A regular HTML `<form>` needs to be defined.

④ Adds a `<appNG:formElement>` for the user's email address (attributes explained below).

⑤ Nest the regular HTML `<input>`-tag inside `<appNG:formElement>`.

⑥ The `<appNG:formConfirmation>` defines which `org.appng.api.FormProcessProvider` to call. The `method`-attributes names the respective Spring bean. The `mode` controls whether the content of this tag should always be shown (`always`) or only if the form has been submitted (`submitted`).

⑦ Adds some parameters that are being passed as a `java.util.Map<String,Object>` to the `process()` -method of the respective `FormProcessProvider` implementation.

The `FormProcessProvider` is only being called if all mandatory fields have a valid value and all rules are satisfied.

The `<appNG:formElement>` is used as a wrapper around standard HTML form input fields, which are

- `<input>` (types: `text`, `radio`, `password`, `hidden`, `file`, `checkbox`)
- `<textarea>`
- `<select>` with nested `<option>`

Attributes: (all optional)

- `mandatory` - set to true if the field is mandatory
- `mandatoryMessage` - the message to be displayed when no value has been entered for a mandatory field
- `errorMessage` - the error message to be displayed when validation fails
- `errorClass` - the CSS class to add to the input field when validation fails
- `errorElementId` - the id of an element to append a with the error message
- `rule` - a validation rule for the input field
- `desc` - a description for the input field

Rules:

Here's a list of the possible values for the `rule` attribute.

Name	Description	Example
`string`	only word characters ([a-zA-Z_0-9] allowed)	`rule="string"`
`email`	must be a valid email address	`rule="email"`
`equals`	must be equal to another field or value	`rule="equals('foo')"` `rule="equals(anotherfield)"`
`regExp`	must match the given regular expression	`rule="regExp('[A-F0-9]+')"`
`number`	must be a number	`rule="number"`
`numberFractionDigits`	must be a number with up to x digits, and y fractional digits	`rule="number(2,4)"`
`size`	must have an exact length of x	`rule="size(3)"`
`sizeMin`	must have a minimum length of x	`rule="sizeMin(3)"`
`sizeMax`	must have a maximum length of x	`rule="sizeMax(3)"`
`sizeMinMax`	must have a minimum length of x and a maximum length of y	`rule="sizeMinMax(3,5)"`
`fileType`	must have one of the comma-separated types (`type="file"` only)	`rule="fileType('tif,pdf')"`
`fileSizeMin`	must have a minimum size of x MB/KB (`type="file"` only)	`rule="fileSizeMin('0.5MB')"`
`fileSizeMax`	must have a maximum size of x MB/KB (`type="file"` only)	`rule="fileSizeMax('5.0MB')"`
`fileSize`	must have a size between x and y MB/KB (`type="file"` only)	`rule="fileSize('500KB','5.0MB')"`

Name	Description	Example
`fileCount`	between x and y files must have been selected (`type="file"` only)	`rule="fileCount(1,10)"`
`fileCountMin`	at least x files must have been selected (`type="file"` only)	`rule="fileCountMin(5)"`
`fileCountMax`	at most x files must have been selected (`type="file"` only)	`rule="fileCountMax(5)"`
`captcha`	Must match a captcha value. The result of the captcha is stored in the variable `SESSION['SESSION']['captcha']`, where the first `SESSION` means the HTTP Session, `['SESSION']` the name of an attribute within the HTTP session. Since this attribute is also a map, you can use `['captcha']` to retrieve the result.	`rule="captcha(SESSION['SESSION']['captcha'])"`

Make sure you also check the Javadoc of the form tags:

- org.appng.taglib.form.Form
- org.appng.taglib.form.FormData
- org.appng.taglib.form.FormElement
- org.appng.taglib.form.FormGroup
- org.appng.taglib.form.FormConfirmation

10.2.1. <appNG:formGroup>

A HTML `<select>` and a group of radio buttons (`<input type="radio">`) must be wrapped by an `<appNG:formGroup>`.

Example for a `<select>`:

```
<appNG:formGroup name="subject" ①
        mandatory="true" mandatoryMessage="This field is mandatory!"  ②
        errorClass="error" errorElementId="subject_error">  ③
    <select name="subject"> ④
        <appNG:formElement> ⑤
            <option value="">Please select</option>
        </appNG:formElement>
        <appNG:formElement>
            <option value="A">Option A</option>
        </appNG:formElement>
        <appNG:formElement>
            <option value="B">Option B</option>
        </appNG:formElement>
            <appNG:formElement><option value="C">Option C</option>
        </appNG:formElement>
    </select>
    <div id="subject_error"></div> ⑥
</appNG:formGroup>
```

① Defines the name of the group.

② Make the field mandatory and provide a message.

③ Define the class and the id of the element holding the error message.

④ Use the group's name also as name for the `<select>`.

⑤ Add a `<appNG:formelement>` for each option.

⑥ The element holding the error message, if any.

A `<appNG:formGroup>` is rendered as a `<div>`, which -in case of an error- also receives the `errorClass`.

Example for a radio group:

```
<appNG:formGroup name="gender"
        mandatory="true" mandatoryMessage="This field is mandatory!"
        errorClass="invalid"  errorElementId="gender_error">
        <appNG:formElement>
            <input type="radio" value="M" name="gender"><label>Male</label>
        </appNG:formElement>
        <br/>
        <appNG:formElement>
            <input type="radio" value="F" name="gender"><label>Female</label>
        </appNG:formElement>
    </select>
    <div id="gender_error" class="error"></div>
</appNG:formGroup>
```

10.3. Search tags

These tags provide functionality to make use of the indexing and searching-features of appNG. It must contain at least one `<appNG:searchPart>` tag.

See the chapter about Indexing and Searching for details on this topic.

10.3.1. <appNG:search>

This tag is used to retrieve the search results in a certain format. The 'json'-format is useful if some Javascript renders the results, while 'xml' can be used to apply a XSL stylesheet to the results.

Attributes: (defaults in braces)

- `format` ('json')- one of `xml` or `json`
- `parts` (`false`) - whether the resulting XML/JSON should be split in parts
- `highlight` (`span`) - the x(ht)ml-tag used to highlight the search term within the search results.

Parameters: (defaults in braces)

- `pageSize` (`25`)
  the page size to use
- `pageSizeParam` (`pageSize`)
  the name of the request parameter that contains the page-size
- `pageParam` (`page`)
  the name of the request parameter that contains the current page
- `queryParam` (`q`)
  the name of the request parameter that contains the search term
- `maxTextLength` (`150`)
  the maximum length of a search result text
- `dateFormat` (`yyyy-MM-dd`)
  the date pattern used to format dates
- `fillWith` (`…`)
  the placeholder used when the search result text is being stripped
- `xsl`
  the path to the XSLT stylesheet to use when format is XML
- `pretty` (`false`)
  if the XML/JSON output should be formatted prettily

Example:

```
<appNG:search parts="false" format="json" highlight="span">①
        <appNG:param name="queryParam">term</appNG:param>
        <appNG:param name="pageSize">10</appNG:param>
        <appNG:param name="pageParam">p</appNG:param>②
        ③
        <appNG:searchPart
            application="global"
            language="de"
            title="Search Results"
            fields="title,contents" analyzerClass=
"org.apache.lucene.analysis.de.GermanAnalyzer"/>
 </appNG:search>
```

① use `json` format

② set some parameters by overriding their default

③ add a `<appng:searchPart>` with `application="global"`, meaning instead of calling specific application, the results from the standard global search should be used

Predefined fields

The following field are predefined by appNG and should not be misused by putting different kind of information into them:

- `path` - then path to the document, relative to the site's domain
- `title` - the title of the document
- `teaser` - a short intro text for the document
- `image` - an image for the document
- `date` - the date of the last change, using the pattern `yyyy-MM-dd HH:mm:ss`
- `type` - the type of the document
- `language` - the language of the document
- `contents` - the textual content of the document
- `id` - the Id of the document

10.3.2. <appNG:searchPart>

As shown above, an `<appNG:search>` can contain different `<appNG:seachPart>`-elements. An `<appNG:seachPart>` refers to an implementation of org.appng.search.SearchProvider.

Attributes:

- `application` - the application that provides the `SearchProvider`
- `method` - the name of the Spring bean implementing `SearchProvider`
- `language` - the language of the documents to find
- `title` - the title for this part

- `fields` - a comma-separated list of the document's field to search in
- `analyzerClass` - the class implementing org.apache.lucene.analysis.Analyzer, used when performing the search

Parameters: Any parameter recognized by the `SearchProvider`.

Example:

```
<appNG:searchPart
        application="acme-products"
        method="productSearchProvider"
        language="en"
        title="ACME Products"
        fields="title,contents"
        analyzerClass="org.apache.lucene.analysis.en.EnglishAnalyzer">
    <appNG:param name="foo">bar</appNG:param>
    <appNG:param name="jin">fizz</appNG:param>
</appNG:searchPart>
```

10.3.3. <appNG:searchable>

This tag is used to mark certain areas of a JSP page as searchable, meaning those parts should be added as a document to the site's search index.

Attributes:

- `index` - whether the body content should be indexed
- `visible` (`true`) - whether or not the body content should be displayed
- `field` - the name of the field. May be one of the standard fields (`title`, `teaser`, `image`, `contents`) or a user-defined one.

For JSPs, the fields `id`, `date`, `type`, `language` and `path` are set by appNG and thus must not be set with `<appNG:searchable>`.

Example:

```
①
<appNG:searchable index="true" field="title" visible="false">
    The Hitchhiker's Guide to the Galaxy
</appNG:searchable>
②
<appNG:searchable index="true" field="contents">
The Hitchhiker's Guide to the Galaxy is a comic science fiction series created by
Douglas Adams.
    ③
    <appNG:searchable index="false">
        This text is being ignored.
    </appNG:searchable>
The title is the name of a fictional, eccentric, electronic travel guide, The
Hitchhiker's Guide to the Galaxy, prominently featured in the series.
    </appNG:searchable>
④
<appNG:searchable index="true" field="customfield" visible="false">
    Hitchhiker
</appNG:searchable>
```

① define an invisible field for the `title`

② define the `contents`-field (multiple occurences are allowed!)

③ tags can be nested, if some areas should be excluded (use `index="false"` in those cases)

④ adds an invisible field named `customfield`

Excluding a page from being indexed

The `<appNG:searchable>` tag can also be used the **exclude** a page from being indexed. Therefore, use the field `indexPage` and set it's value to `false`. No other attributes are required.

```
<appNG:searchable field="indexPage">false</appNG:searchable>
```

As an alternative, you can use a surrounding `<appNG:searchable index="false">` tag.

```
<appNG:searchable index="false">
    <appNG:searchable index="true" field="title">Title</appNG:searchable>
    <appNG:searchable index="true" field="contents">Contents</appNG:searchable>
</appNG:searchable>
```

10.4. Other tags

10.4.1. <appNG:param>

The `<appng:param>` tag can be used to add parameters to a

`<appNG:taglet>`

- <appNG:search>
- <appNG:searchPart>
- <appNG:formConfirmation>
- <appNG:application>

Attributes:

- name - the name of the parameter
- unescape - if set to true, HTML entities in the value are unescaped before passed to the owning tag

For the parameter's value, you can reference to request parameters using the syntax #[<param>].

Example:

```
<appNG:param name="replyTo">#[email]</appNG:param>
```

10.4.2. <appNG:attribute>

With this tag, you can read/ write attributes of different Scopes from/ to the current environment.

Attributes:

- scope - the scope of the attribute (REQUEST, SESSION, PLATFORM or URL)
- mode - the mode (read or write)
- name - the name of the attribute
- value - write mode only: the value to write

Example:

```
<appNG:attribute scope="SESSION" mode="read"  name="foo"  />
<appNG:attribute scope="REQUEST" mode="read"  name="bar" />
<appNG:attribute scope="SESSION" mode="write" name="someName" value="someValue"/>
<!--
For URL-scope, the name is the zero based index of the path segment
(segments are separated by '/'). For example, if the path is
'/en/foo/bar/42' then you can access the '42' with index 3
-->
<appNG:attribute scope="URL" mode="read" name="3" />
```

Only the REQUEST and SESSION scope allow the write mode.

10.4.3. <appNG:if>

This tag is used to display the tag body only if the given condition matches. In the condition, any request parameter can be used. Expressions are supported, but no leading ${ and closing } is

required.

Attributes:

- `condition` - the condition to be satisfied

Example:

```
<!--assumed there's a request parameter named 'foo' -->
<appNG:if condition="foo eq 'bar'">Foobar!</appNG:if>
```

10.4.4. <appNG:permission>

If you want to make sure a that certain content is only visible if the logged-in user has a certain permission, use the `<appNG:permission>` tag.

Attributes:

- `application` - the name of the application that provides the permission
- `permission` - the name of the permission

Example:

```
<appNG:permission application="myapp" permission="showSecretContent">
This is secret content!
</appNG:permission>
```

10.5. <appNG:application>

Used to embed an appNG application inside a JSP page. This is achieved by transforming the `<platform>` XML document returned by the application with a custom XSL stylesheet. Thus it is possible to adjust the appearance of the application to any required design.

Attributes:

- `application` - the name of the application that should be embedded.

Parameters: (`<appNG:param>`)

- `defaultBaseUrl`
  The url of the page where the application is embedded
- `defaultPage`
  The default page of the application, used when the url parameters do not contain a page name
- `xslStyleSheet`
  The path to the XSLT stylesheet used for transformation, relative to the site's repository folder. If omitted, the plain XML is written as an HTML comment.

- `requestAttribute`
  The name of an environment-attribute with the scope `REQUEST` where the transformation result should be stored in. If this parameter is not set, the result is directly written to the page.

GET-Parameters:

- `xsl` - if `false`, the plain XML is written as an HTML comment

Example:

The following example assumes you want to embed the application `acme-app` into the page `/en/acme`. Every path segment after `/en/acme` is passed as an url-parameter to the application.

```
<appNG:application name="acme-app">
  <appNG:param name="defaultBaseUrl">/en/acme</appNG:param>
  <appNG:param name="defaultPage">/index/welcome</appNG:param>
  <appNG:param name="xslStyleSheet">/meta/xsl/acme/platform.xsl</appNG:param>
  <appNG:param name="requestAttribute">acmeResult</appNG:param>
</appNG:application>
<!-- later in JSP -->
<appNG:attribute mode="read" name="acmeResult" scope="REQUEST" />
```

11. Indexing and Searching

Each appNG site owns its own document based index. Documents are added to/ retrieved from the index in the form of an org.appng.api.search.Document. Each site has its own indexing thread, that is waiting for org.appng.api.search.DocumentProducers to offer some org.appng.api.search.DocumentEvents

11.1. Adding documents at runtime

An application can provide a `org.appng.api.search.DocumentProducer` with these steps:

1. Retrieve a org.appng.api.model.FeatureProvider by calling Application.getFeatureProvider()
2. Call FeatureProvider.getIndexer() to obtain a `Consumer< DocumentEvent, DocumentProducer>`
3. Add a DocumentProducer to this `Consumer<DocumentEvent,DocumentProducer>`
4. Provide org.appng.api.search.Documents by calling `DocumentProducer.put(DocumentEvent e)`

Example:

```
import org.appng.api.search.Consumer;
import org.appng.api.search.Document;
import org.appng.api.search.DocumentEvent;
import org.appng.api.search.DocumentProducer;
...
Consumer<DocumentEvent, DocumentProducer> indexer =
    application.getFeatureProvider().getIndexer();①
DocumentProducer documentProducer =
    new DocumentProducer(EnglishAnalyzer.class, "myapp-indexer");②
indexer.put(documentProducer);③
Document docAdded = ... // ④
documentProducer.put(new DocumentEvent(docAdded, Document.CREATE)); ⑤
```

① retrieve the `indexer`

② create a new `DocumentProducer`

③ add the `producer` to the `indexer`

④ create a `Document`, you may extend org.appng.search.indexer.SimpleDocument here

⑤ add a new `DocumentEvent` to the producer

11.2. Adding Documents at the time of indexing

If the appNG site has the appNG scheduler application enabled (which will be the case in most scenarios) there's the org.appng.search.DocumentProvider-interface that can be implemented by the application to provide an `java.lang.Iterable` of org.appng.api.search.DocumentProducers.

The appNG scheduler application contains an indexing job that automatically detects all the Spring beans implementing `DocumentProvider`.

```
import org.appng.api.search.Document;
import org.appng.api.search.DocumentEvent;
import org.appng.api.search.DocumentProducer;
import org.appng.search.DocumentProvider;

@org.springframework.stereotype.Component ①
public class MyDocumentProvider implements DocumentProvider {

    public Iterable<DocumentProducer>
            getDocumentProducers(Site site, Application application) { ②
        DocumentProducer documentProducer =
                new DocumentProducer(EnglishAnalyzer.class, "myapp-indexer"); ③
        for(/* loop over some domain objects */) {
            Document docAdded = ...; // build document
            documentProducer.put(new DocumentEvent(docAdded, Document.CREATE)); ④
        }
        return java.util.Arrays.asList(documentProducer);
    }
}
```

① `MyDocumentProvider` is a `@Component`, so the indexing job can detect it.

② The single method to implement is `getDocumentProducers(Site site, Application application)`.

③ Create a `DocumentProducer` by providing the Analyzer to use and a name.

④ Since the indexing-job is completely rebuilding the index, we use the event type Document.CREATE.

11.3. Adding documents through `<appNG:searchable>`

The `<appNG:searchable>` JSP-tag can be used to add documents to the index when a site's JSP repository is being indexed by the appNG scheduler application. Details about the `<appNG:searchable>`-tag can be found in the section about search tags.

11.4. Adding documents at the time of searching

Instead of providing documents at indexing time, an application can also provide documents right a the time of searching. This can be achieved by implementing org.appng.search.SearchProvider.

The `SearchProvider` can then be referenced by a `<appNG:searchPart>`-tag. See the section about search tags for details.

12. Testing

12.1. General

appNG also offers support for unit- and integration testing your appNG applications. Therefore, it

uses the testing capabilities of the Spring framework. See the Reference Documentation for details on testing with Spring.

To enable test support, just add the following dependency to your `pom.xml`:

```
<dependency>
    <groupId>org.appng</groupId>
    <artifactId>appng-testsupport</artifactId>
    <version>${appNG.version}</version>
</dependency>
```

Next, use org.appng.testsupport.TestBase as a base class for your unit tests. You can test with or without JPA support enabled.

Without JPA support:

```
@org.springframework.test.context.ContextConfiguration(initializers = EmployeesTest
.class)
public class EmployeesTest extends TestBase {

    public EmployeesTest() {
        super("myapp", "application-home");
    }

}
```

With JPA support:

```
@org.springframework.test.context.ContextConfiguration(
    locations = { TestBase.TESTCONTEXT_JPA }, initializers = EmployeesTest.class)
public class EmployeesTest extends TestBase {

    public EmployeesTest() {
        super("myapp", "application-home");
        setEntityPackage("com.myapp.domain");
        setRepositoryBase("com.myapp.repository");
    }

}
```

12.2. Testing a datasource

```
@org.junit.Test
public void testShowEmployees() throws ProcessingException, IOException {
    addParameter("selectedId", "1"); ①
    initParameters(); ②
    DataSourceCall dataSourceCall = getDataSource("employees");③
    CallableDataSource callableDataSource = dataSourceCall.getCallableDataSource(); ④
    callableDataSource.perform("aPage"); ⑤
    validate(callableDataSource.getDatasource());⑥
}
```

① adds a parameter

② initialize the parameters, must be called after parameters have been added

③ retrieve a `DataSourceCall` representing the datasource by its id

④ get a `CallableDataSource`

⑤ perform the `CallableDataSource`

⑥ validate the response

In step 6, a org.appng.testsupport.validation.WritingXmlValidator is used to compare the contents of a **control file** with the XML that results from marshalling the given object (in this case a `org.appng.xml.platform.Datasource`). The control file must be located at `src/test/resources/xml`. It's name is derived from the name of the test class and the name of the test method. So in this example, it would be `EmployeesTest-testShowEmployees.xml`.

For initially creating and later updating your control files, just set `WritingXmlValidator.writeXml = true` and the control files will be written to `src/test/resources/xml`.

12.3. Testing an action

```
@org.junit.Test
public void testCreateEmployee() throws ProcessingException, IOException {
    ActionCall action = getAction("employeeEvent", "create");①
    action.withParam(FORM_ACTION, "create");②
    Employee formBean = new Employee("John", "Doe");③
    CallableAction callableAction = action.getCallableAction(formBean);④
    FieldProcessor fieldProcessor = callableAction.perform();⑤
    validate(fieldProcessor.getMessages(), "-messages");⑥
    validate(callableAction.getAction(), "-action");⑦
}
```

① retrieve an `ActionCall` representing the action by its event-id and id

② add required parameters to the action

③ create a form bean

④ retrieve a `CallableAction`

⑤ perform the `CallableAction` receive a `FieldProcessor`

⑥ use a `validate(…)`-method that takes a suffix as a parameter, validating the messages of the `fieldProcessor`

⑦ use the same validate method to validte the contents of the action

Although you pass the `formBean` as a whole to `ActionCall.getCallableAction(formBean)`, this `formBean` is being copied internally. This copy, which is passed to the `ActionProvider<T>`, will contain only those properties that are mapped and writable in the datasource used by the action.

12.4. Adding custom bean definitions for testing

You can add custom bean definitons for your tests. For example, if you want to run a SQL script to initialize your test database, you could provide a file `beans-test-xml` located at `src/test/resources`.

beans-test.xml

```
<?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:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jdbc:initialize-database enabled="true" data-source="datasource">①
        <jdbc:script location="classpath:/sql/init-db.sql" />②
    </jdbc:initialize-database>
</beans>
```

① use `<jdbc:initialize-database>` and make a reference to the built in datasource

② set the classpath location for the DDL script

In your testcase, you just need to add `beans-test.xml` to the `@ContextConfiguration`-annotation.

```
@ContextConfiguration(locations = { TestBase.TESTCONTEXT_JPA, "classpath:/beans-
test.xml" }
```

12.5. Test utilities

appNG comes along with some handy test utilities.

12.5.1. Writing JSON Validator

The class org.appng.testsupport.validation.WritingJsonValidator is a utility to map Java objects to

JSON and compare it against a reference file. It also provides the functionality to easily create the reference files at `src/test/resources/json` in your project.

The folder for reference files must be created manually before creating reference files.

```
public class MyTest {

  static { ①
    WritingJsonValidator.writeJSON = false; ②
    WritingJsonValidator.sortPropertiesAlphabetically = true; ③
  }

  @Test
  public void testWithJSON(){
    ComplexObject co = new ObjectProviderUnderTest().getObject();
    WritingJsonValidator.validate(co,"/filename.json"); ④
  }

}
```

① It is handy to configure general properties in a static block for all test methods of a test case.

② If the reference file does not exist or is outdated, set this property to `true` to create a new reference file.

③ The order of object properties in a JSON file is not clearly defined. Set this property to `true` for an alphabetically ordered output.

④ Call the `validate`-method of WritingJsonValidator with the object and the file name of the reference file to map the object to JSON and compare it against the JSON provided in the compare file.

In Eclipse, a newly created reference file might not be available immediately after creation. The project must be refreshed first.

13. Implementing services

13.1. Webservices

A webservice is the suitable solution if your application needs to provide

- images, probably scaled and otherwise edited using an ImageProcessor
- file attachments (aka downloads)
- custom formatted data as JSON, HTML, XML or any other required format

The interface to implement is org.appng.api.Webservice or org.appng.api.AttachmentWebservice.

Example:

```
@org.springframework.stereotype.Service
public class ImageService implements Webservice {

  public byte[] processRequest(Site site, Application application,
      Environment environment, Request request) throws BusinessException {
      Integer id = request.convert(request.getParameter("id"), Integer.class);
      Image image = getImage(id)
      return image.getData(id);
  }

  public String getContentType() {
    return "image/jpg";
  }

}
```

The URL schema for a webservice is

`http(s)://<host>[:<port>]/service/<site-name>/<application-name>/webservice/<service-name>`

Example: http://localhost:8080/service/manager/myapp/webservice/myWebservice

13.2. SOAP services

The interface org.appng.api.SoapService is used to implement a SOAP compliant service based on Spring WS. For more information on Spring WS, see the Reference Documentation and this Getting Started Guide.

The following steps are required to build a SOAP service:

1. Define a XSD schema

   Because Spring WS uses the contract-first approach, it is essential to have an XSD schema defining the structure of the request and response elements. The schema file should be placed in `src/main/xsd`.

   A minimalist XSD looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
"http://example.com/soap">①
    <xs:element name="exampleMethodRequest">②
        <xs:complexType>
            <xs:sequence>
                <xs:element name="requestParam" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="exampleMethodResponse">②
        <xs:complexType>
            <xs:sequence>
                <xs:element name="responseParam" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

① define the namespace for the schema

② define the request and response elements, following the naming convention `xyzRequest` and `xyzResponse`

2. Generate JAXB classes for the schema file

   The jaxb2-maven-plugin can be used to generate the Java classes using the xjc binding compiler on this schema. The build-helper-maven-plugin is then used to add those classes to the project sources.

   Example configuration from `pom.xml`:

```
<build>
  <resources>
    <resource>
      <directory>src/main/xsd</directory>①
    </resource>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jaxb2-maven-plugin</artifactId>
      <version>2.2</version>
      <executions>
        <execution>
          <id>generate</id>
          <goals>
            <goal>xjc</goal>
          </goals>
          <phase>generate-sources</phase>
          <configuration>
            <packageName>com.example.soap</packageName>②
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <version>1.12</version>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
            <sources>
              <source>target/generated-sources/jaxb</source>③
            </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

① add `src/main/xsd` to the project's `resources`

② generate JAXB classes in the package `com.example.soap`

③ add the folder with the generated JAXB classes as a source directory

3. Implement the `SoapService`

```
import org.appng.api.Environment;
import org.appng.api.SoapService;
import org.appng.api.model.Application;
import org.appng.api.model.Site;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;

import com.example.soap.ExampleMethodRequest;
import com.example.soap.ExampleMethodResponse;

@Endpoint ①
public class ExampleSoapService implements SoapService { ②

  private Site site;
  private Application application;
  private Environment environment;

  ③
  @PayloadRoot(namespace = "http://example.com/soap", localPart =
"exampleMethodRequest")
  ④
  public @ResponsePayload ExampleMethodResponse getExample(
        @RequestPayload ExampleMethodRequest request) {
    ExampleMethodResponse exampleMethodResponse = new ExampleMethodResponse();
    exampleMethodResponse.setResponseParam(request.getRequestParam());
    return exampleMethodResponse;
  }

  ⑤
  public String getSchemaLocation() {
    return "example.xsd";
  }

  ⑥
  public String getContextPath() {
    return "com.example.soap";
  }

  // getters and setters here

}
```

① annotate with `@org.springframework.ws.server.endpoint.annotation.Endpoint`

② implement `org.appng.api.SoapService`

③ define a `@PayloadRoot` that uses the `namespace` of the XSD schema and defining the `localPart`

④ the method must use the a request-object as an `@RequestPayload` argument and response-object as an `@ResponsePayload` return type

⑤ return the classpath-relative location of the schema file

⑥ return the package name of the generated JAXB classes

The URL schema for the WSDL of the service is

`http(s)://<host>[:<port>]/service/<site-name>/<application-name>/soap/<service-name>/<service-name>.wsdl`

Example: http://localhost:8080/service/manager/myapp/soap/exampleSoapService/exampleSoapService.wsdl

13.3. REST services

With appNG, you can easily create REST-based services by using the Spring framework's designated annotations. The most important ones to use here are @RestController, @RequestMapping and @PathVariable.

Let's implement the add-operation of a simple `CalculatorService` as a `@RestController`.

```
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController ①
public class CalculatorService {

    @RequestMapping(value = "/add/{a}/{b}", ②
        method = RequestMethod.GET, ③
        produces = MediaType.TEXT_PLAIN_VALUE) ④
    public ResponseEntity<Integer> add( ⑤
            @PathVariable("a") Integer a, @PathVariable("b") Integer b) { ⑥
        return new ResponseEntity<Integer>(a + b, HttpStatus.OK); ⑦
    }
}
```

① Define the class ass `@RestController`

② Define a `@RequestMapping` and the path it should match, using path variables

③ Define the HTTP method

④ Define the media-type this method produces

⑤ Return a `ResponseEntity` of the desired type

⑥ Map the path variables to the parameters

⑦ Return the response

In order to make the necessary conversion of the `ResponseEntity` work, there needs to be an appropriate HttpMessageConverter present in your application.

For the example above, this converter looks like this:

```
@Service
public class IntegerMessageConverter implements HttpMessageConverter<Integer> {

    public List<MediaType> getSupportedMediaTypes() {
        return Arrays.asList(MediaType.TEXT_PLAIN);
    }

    public boolean canWrite(Class<?> clazz, MediaType mediaType) {
        return Integer.class.equals(clazz);
    }

    public void write(Integer t, MediaType contentType, HttpOutputMessage
outputMessage)
            throws IOException, HttpMessageNotWritableException {
        outputMessage.getBody().write(String.valueOf(t).getBytes());
    }

    public boolean canRead(Class<?> clazz, MediaType mediaType) {
        return false;
    }

    public Integer read(Class<? extends Integer> clazz, HttpInputMessage inputMessage)
            throws IOException, HttpMessageNotReadableException {
        return null;
    }

}
```

If your request-/response-types are JAXB-generated classes, you have to use a MarshallingHttpMessageConverter instead.

For JSON-format, there's a MappingJackson2HttpMessageConverter available.

The URL schema for the REST service is

`http(s)://<host>[:<port>]/service/<site-name>/<application-name>/rest/path/to/service`

Example: http://localhost:8080/service/manager/myapp/rest/add/3/4

13.3.1. Exception handling

For handling exceptions, you can (and should) use Spring's @ExceptionHandler. You may also use a bean annotated with @ControllerAdvice for centralized exceptiong handling. More details can be found in the corresponding section of Spring's reference documentation.

A method annotated with `@ExceptionHandler` can use a Site, Application and Environment as a parameter.

13.3.2. JSON REST services

```
/*
 * Copyright 2011-2018 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.example;

import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CalculatorService {

    @RequestMapping(value = "/add", ①
            method = RequestMethod.POST, ②
            produces = MediaType.APPLICATION_JSON_UTF8_VALUE) ③
    public ResponseEntity<Result> add(@RequestBody Operators operators) { ④
        return new ResponseEntity<Result>(new Result(operators.a, operators.b),
HttpStatus.OK);
    }

    class Operators {
        Integer a;
        Integer b;
```

```
        public Integer getA() {
            return a;
        }

        public void setA(Integer a) {
            this.a = a;
        }

        public Integer getB() {
            return b;
        }

        public void setB(Integer b) {
            this.b = b;
        }
    }

    class Result extends Operators {
        Integer result;

        public Result(Integer a, Integer b) {
            this.a = a;
            this.b = b;
            this.result = a + b;
        }

        public Integer getResult() {
            return result;
        }

        public void setResult(Integer result) {
            this.result = result;
        }
    }

}
```

① Map to the path `/add`

② Only allow the `POST` method

③ Use constant from MediaType to set the returned content type

④ Use an `Operators` object as a parameter and annotate it with RequestBody.

A method annotated with `@RequestMapping` can use appNG's Site, Application and Environment interfaces as a parameter.

Also note that there are many built-in HandlerMethodArgumentResolvers that add support for additional parameter types.

To support JSON format, you have to register the aforementioned `MappingJackson2HttpMessageConverter` in your `beans.xml`:

```
<bean class=
"org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
  <property name="prettyPrint" value="true" /> ①
</bean>
```

① Use `prettyPrint` for a better readability.

The **request** will use the `POST` method and JSON formatting for it's content:

```
{
  "a" : 39,
  "b" : 3
}
```

The **response** to this request looks like this:

```
{
  "a" : 39,
  "b" : 3,
  "result" : 42
}
```

For more details on this topic, see section 22.3.3 of Spring's reference documentation.

Multipart requests

When using multipart requests (`enctype="multipart/form-data"`) in a `@RestController`, it is not possible to use a MultipartFile as a parameter. The reason for that is the fact that appNG parses the request by itself and wraps it in a org.appng.forms.Request.

You can access this interal `org.appng.forms.Request` to retrieve the uploaded files as shown below (the `httpServletRequest` is a parameter of a RESTful method):

```
@PostMapping("/file-upload")
public ResponseEntity<String> handleFileUpload(HttpServletRequest httpServletRequest)
{
    org.appng.forms.Request request = (org.appng.forms.Request)
        httpServletRequest.getAttribute(org.appng.forms.Request.REQUEST_PARSED);
    List<org.appng.forms.FormUpload> formUploads = request.getFormUploads("file");
  ...
}
```

As an alternative, you can just implement a Webservice and use RequestContainer.getFormUploads() to retrieve the uploaded files.

13.4. Job scheduling

A org.appng.api.ScheduledJob is a (periodically or manually triggered) task that can be defined by an application. The implementation is simple:

```
import java.util.Map;

import org.appng.api.ScheduledJob;
import org.appng.api.model.Application;
import org.appng.api.model.Site;

public class DemoJob implements ScheduledJob {

    private String description;
    private Map<String, Object> jobDataMap;

    public void execute(Site site, Application application) throws Exception {
        // do something
    }

    // getters and setters here

}
```

With the `jobDataMap`, the required configuration parameters can be passed to the job. There are some predefined parameters for a job:

- `cronExpression`
  A cron-expression describing when the job should be executed. The Cronmaker is a useful tool helping you to build those expressions.
- `enabled`
  If set to `true` and the `cronExpression` property is properly set, the job will automatically be scheduled when the appNG platform starts.
- `runOnce`
  If set to `true`, this job will only run once per appNG cluster, i.e. it is not executed on each node, which is the default behavior.
- `hardInterruptable`
  If set to `true`, this job can safely be interrupted, e.g when a `Site` is being reloaded. This is achieved by running the job in a separate thread and calling Thread.interrupt.
- `allowConcurrentExecutions`
  If set to `true`, multiple instances of this job can run concurrently (default is `false`).

The best way to configure a job is in `beans.xml`:

```
<bean id="demoJob" class="com.myapp.job.Demojob">
  <property name="jobDataMap">
    <map>
      <entry key="enabled" value="true" />
      <entry key="runOnce" value="true" />
      <entry key="cronExpression" value="0 0 8 ? * *" />①
      <entry key="foo" value="bar" />②
      <entry key="answer" value="42" />
    </map>
  </property>
</bean>
```

① add the standard predefined parameters

② add some custom parameters

14. Commonly used API

As you may have noticed, there are a handful of interfaces that occur quite often when implementing functionality with appNG:

- org.appng.api.model.Site
- org.appng.api.model.Application
- org.appng.api.Request
- org.appng.api.Environment
- org.appng.api.model.Properties

The following section explains those in more detail.

14.1. `org.appng.api.model.Site`

A Site first and foremost is used for aggregating several applications under a corporate domain. Most frequently, `site.getProperties()` is used to retrieve the site's configuration in order to build some (relative or absolute) links, for example to a `WebService`:

The constants for the available property names can be found at org.appng.api.SiteProperties.

```
String servicePath = site.getProperties().getString(SiteProperties.SERVICE_PATH);
String absolutePath = String.format("%s/%s/%s/%s/webservice/pictureService?id=4711",
site.getDomain(), servicePath, site.getName(), application.getName()));
```

Another common use-case for a site is to redirect the user to another location using `site.sendRedirect()`.

14.2. org.appng.api.model.Application

Through the Application interface, any relevant information about the current application can be retrieved. This includes

- Properties (`getProperties()`)
- Resources (`getResources()`)
- Permissions (`getPermissions()`)
- Roles (`getRoles()`)
- ApplicationSubjects (`getApplicationSubjects()`)
- FeatureProvider (`getFeatureProvider()`)

Please check the corresponding JavaDoc for details.

14.3. org.appng.api.model.Request

The Request offers a large number of methods which serve one of the following purposes:

- retrieving request values (`getParameter()`, `getParameterList()`, `getParameterNames()`, `getParameters()`, `getParametersList()`, `getFormUploads()`)
- validating objects (`validateBean()`, `validateField()`)
- binding data (`fillBindObject()`, `setPropertyValue()`, `setPropertyValues()`)
- converting values (`canConvert()`, `convert()`)
- internationalization (`getMessage()`, `getMessageSource()`)
- retrieving information about the environment (`getEnvironment()`, `getSubject()`, `getLocale()`)

Please check the corresponding JavaDoc for details.

14.4. org.appng.api.Environment

The Environment is used for storing and retrieving differently scoped attributes using `getAttribute(Scope scope, String name)` and `setAttribute(Scope scope, String name, Object value)`.

Below you can find a description of the available org.appng.api.Scopes.

14.4.1. PLATFORM scope

This scope is mainly used for exchanging platform related configuration across different core components and is bound to the javax.servlet.ServletContext appNG is executed in. Therefore, this scope is only accessible from the appNG core and **privileged** applications.

The constants used as attributes names can be found at org.appng.api.Platform.Environment.

Example usage:

```
Map<String, Site> siteMap = environment.getAttribute(Scope.PLATFORM, Platform
.Environment.SITES);
```

Even for privileged applications, the `PLATFORM` scope should be considered as 'read only', because it contains data that is critical for executing appNG correctly.

14.4.2. `SITE` scope

The `SITE` scope can be used to store and retrieve data that is **not** bound to a particular session and thus needs to be available on a more global level. As the `PLATFORM` scope, it is bound to the javax.servlet.ServletContext appNG is executed in. There are no built-in attributes that are stored in the `SITE` scope.

14.4.3. `SESSION` scope

This is propably the most commonly used scope when developing appNG applications. It is bound to the javax.servlet.http.HttpSession and therefore keeps track of the user's session state. For example, you can use it to store the current workflow state within your application.

The built-in constants used as attribute names can be found at org.appng.api.Session.

Example usage:

```
String sessionId = environment.getAttribute(Scope.SESSION, Session.SID);
MyApplicationState state = environment.getAttribute(Scope.SESSION, "
myApplicationState");
if ( null == state ) {
  state = new MyApplicationState();
  environment.setAttribute(Scope.SESSION, "myApplicationState", state);
}
```

14.4.4. `REQUEST` scope

The `REQUEST` scope is bound to the javax.servlet.ServletRequest and therefore can be used to track the state of the request. If, at all, you will most likely access this scope in a reading way.

The built-in constants used as attribute names can be found at org.appng.api.support.environment.EnvironmentKeys

Example usage:

```
Path path = environment.getAttribute(Scope.REQUEST, EnvironmentKeys.PATH_INFO);
String servletPath = environment.getAttribute(Scope.REQUEST, EnvironmentKeys
.SERVLETPATH);
String queryString = environment.getAttribute(Scope.REQUEST, EnvironmentKeys
.QUERY_STRING);
```

14.4.5. URL scope

The URL scope has solely the purpose of accessing URL parameters in a JSP with the appNG:attribute tag.

14.5. org.appng.api.model.Properties

To configure an Application and a Site, both of these offer the getProperties()-method, returning some org.appng.api.model.Properties. These can be used to retrieve type-safe configuration values, optionally providing a default value.

```
Integer foo = properties.getInteger("foo");
Integer bar = properties.getInteger("bar", 42);
List<String> values = properties.getList("values", "foo;bar", ";");
```

You can also inject single property values into a bean using Spring's @Value annotation:

```
@Value("${foo}")
private String foo;
private String servicePath;

public MyService(@Value("${site.service-path}") servicePath){
  this.servicePath = servicePath;
}
```

Be sure you've read the section about placeholders.

14.6. Sending emails

Using the classes contained in org.appng.mail, it very is easy to send plain text and also HTML mails. First, a MailTransport needs to be defined in beans.xml:

```
<bean id="mailTransport" class="org.appng.mail.impl.DefaultTransport">
  <constructor-arg>
    <props>
      <prop key="mail.smtp.host">example.com</prop>
      <prop key="mail.smtp.port">25</prop>
      <prop key="mail.smtp.starttls.enable">true</prop>
      <prop key="mail.smtp.auth">true</prop>
    </props>
  </constructor-arg>
  <constructor-arg value="user" />
  <constructor-arg value="password" />
</bean>
```

This can than be used to send emails:

```
Mail mail = mailTransport.createMail();
mail.setFrom(mailSender);
mail.setSubject(mailSubject);
mail.addReceiver("johndoe@example.com", RecipientType.TO);
mail.addReceiver("janedoe@example.com", RecipientType.CC);
mail.setTextContent("Hello from appNG");
mail.setHTMLContent("Hello from <strong>appNG</strong>")
mailTransport.send(mail);
```

14.7. Working with images

Often applications are required to resize or crop images, hence appNG offers some tooling for that.

Note that ImageMagick 6.9.9 needs to be installed and be available in the system's path (7.x is currently not supported). Furthermore, the built-in application property `featureImageProcessing` must be enabled.

The following example demonstrates how to fit an image into a (imaginary) box sized 120x120 pixel:

```
FeatureProvider featureProvider = application.getFeatureProvider(); ①
File imageCache = featureProvider.getImageCache(); ②
File sourceFile = new File(imageCache, "tempfile"); ③
byte[] pictureData = getPictureData(); ④
org.apache.commons.io.FileUtils.writeByteArrayToFile(sourceFile, pictureData); ⑤
ImageProcessor imageProcessor = featureProvider.getImageProcessor(sourceFile,
"tempfile-resized"); ⑥
File result = imageProcessor.resize(120, 120).quality(0.8).getImage(); ⑦
```

① Retrieve a `FeatureProvider` from the `Application`.

② Retrieve a folder used for caching the image data.

③ In that folder, create a new file.

④ Retrieve the binary data of the image.

⑤ Write the original image data to the `sourceFile`.

⑥ Retrieve a new ImageProcessor using `sourceFile` as a source and defining the name of the target file (which will be located in the folder `imageCache`).

⑦ Resize the image to make it fit into a (imaginary) box with a size of 120x120 pixel (aspect ratio is kept!), set the quality to 80%, and finally return the target file.

15. Beautifying URLs

When serving JSPs or static content from a site's repository folder, it is often a requirement to rewrite the URLs for some of those contents.

For example, a JSP located at `/en/an-ugly-url` should be available at `/a-beauty-url`.

To solve this problem, appNG uses UrlRewriteFilter. For configuration, the file `urlrewrite.xml` is used, usually located at `/meta/conf/urlrewrite.xml` of a site's root directory.

The configuration file `urlrewrite.xml` is reloaded every minute.
More about the configuration of UrlRewriteFilter can be found in it's user manual.

This file needs to contain a `forward` rule and a `redirect` rule for each URL to be rewritten.

The `urlrewrite.xml` for the above example would look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 4.0//EN"
"http://www.tuckey.org/res/dtds/urlrewrite4.0.dtd">
<urlrewrite>
  <rule> ①
    <from>/en/an-ugly-url</from>
    <to type="redirect" last="true">/a-beauty-url</to>
  </rule>
  <rule> ②
    <from>/a-beauty-url</from>
    <to type="forward" last="true">/en/an-ugly-url</to>
  </rule>
</urlrewrite>
```

① Define a `redirect` rule `<from>` the ugly URL `<to>` the rewritten one.

② Define a `forward` rule `<from>` the rewritten URL `<to>` the ugly one.

The `<from>` element *may* use the caret (`^`) and the dollar sign (`$`) (so called 'anchors' in terms of regular expressions) to mark the beginning and the end of the expression. Either both or none of the must be used.

Note that the order of the rules (from the same type) is essential. Also make sure to use `last="true"` for the `<to>` part of the final rule (of each type), otherwise you may create an endless loop of forwards and redirects.

The `forward` rule is needed for *internally* forwarding the rewritten URL to the real resource. The user will not take notice of this.

The `redirect` rule is needed for actually redirecting the user (a.k.a. browser) from the ugly URL to the desired rewritten URL. Therefore, HTTP status `302` (found) will be used. If HTTP status `301` (moved permanently) is needed, use `type="permanent-redirect"`.

Additionally, the `redirect` rule is used for replacing textual matches of the ugly URL in other JSPs with their rewritten value.

For example, if a JSP contains a `<a href="/en/an-ugly-url">Beautiful!</a>`, the `redirect` rule is

being applied here, resulting in `<a href="/a-beauty-url">Beautiful!</a>`.

The feature of replacing textual matches of the ugly URL with the rewritten one **only** applies to `.jsp` files and **not** to static content (like CSS, HTML JavaScript etc.).

15.1. Dealing with file extensions and GET parameters

If the rewritten resource is a JSP and/or if it must support HTTP GET parameters, the rules need to be enhanced.

Take a look at this example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 4.0//EN"
"http://www.tuckey.org/res/dtds/urlrewrite4.0.dtd">
<urlrewrite use-query-string="true"> ①
  <rule>
    <from>/en/an-ugly-url(\.jsp)?((\?\S+)?)</from> ②
    <to type="redirect" last="true">/a-beauty-url$2</to> ③
  </rule>
  <rule>
    <from>/a-beauty-url((\?\S+)?)</from> ④
    <to type="forward" last="true">/en/an-ugly-url$1</to> ⑤
  </rule>
</urlrewrite>
```

① We need the query string to be part of the `<from>` element, thus we need to set `use-query-string="true"`.

② The ugly URL *may* contain the extension `.jsp`, therefore we add a capturing group `(\.jsp)?`. Also it *may* contain GET parameters, e.g. `?foo=bar&jin=fizz`. For this case, add a capturing group `((\?\S+)?)`.

③ The GET parameters need to be kept, so append them using `$<number-of-capturing-group>`

④ The rewritten URL also *may* contain GET parameters. so capture them with `((\?\S+)?)`.

⑤ Again, append the GET parameters by referring to the capturing group.

As you can see, the rewrite rules do intensely use regular expressions. Therefore, understanding those is essential. A good website for testing your expressions and getting additional explanations is regexr.com.

16. The appNG Maven Archetype

An appNG application uses Apache Maven as a build system, thus the fastest way to setup an appNG application is to use the appNG Maven Archetype as shown below:

```
mvn archetype:generate -DgroupId=mygroupid -DartifactId=myartifactid
-DarchetypeGroupId=org.appng -DarchetypeArtifactId=appng-archetype-application
-DarchetypeVersion=1.18.0-RC1 -DinteractiveMode=false
```

Using this archetype will result in the following `pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>mygroupid</groupId>
    <artifactId>myartifactid</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>myapp</name>
    <description>enter description here</description>

    <properties>
        <outFolder>target</outFolder>
        <appNGVersion>1.18.0-RC1</appNGVersion>
        <displayName>myapplication</displayName>
        <longDescription>enter long description here</longDescription>
        <timestamp>${maven.build.timestamp}</timestamp>
        <maven.build.timestamp.format>yyyyMMdd-HHmm</maven.build.timestamp.format>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.0.0</version>
                <dependencies>
                    <dependency>
                        <groupId>org.appng</groupId>
                        <artifactId>appng-application-assembly</artifactId>
                        <version>${appNGVersion}</version>
                    </dependency>
                </dependencies>
```

```
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>assembly</descriptorRef>
                    </descriptorRefs>
                    <finalName>${project.artifactId}-${project.version}-
${timestamp}</finalName>
                    <appendAssemblyId>false</appendAssemblyId>
                    <outputDirectory>${outFolder}</outputDirectory>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <profiles>
        <!-- a profile for local development -->
        <!-- builds the application directly into the right folder of the appNG
installation -->
        <!-- note that the application must once be installed in file-based mode to
make this work -->
        <profile>
            <id>local</id>
            <properties>
                <outFolder>${env.CATALINA_HOME}/webapps/ROOT/applications/</outFolder>
            </properties>
            <build>
                <plugins>
                    <plugin>
                        <artifactId>maven-assembly-plugin</artifactId>
                        <configuration>
                            <descriptorRefs>
                                <descriptorRef>assembly-local</descriptorRef>
                            </descriptorRefs>
                            <finalName>${project.artifactId}</finalName>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
        </profile>
    </profiles>

    <dependencies>
        <dependency>
            <groupId>org.appng</groupId>
```

```
            <artifactId>appng-api</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.appng</groupId>
            <artifactId>appng-testsupport</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.appng</groupId>
                <artifactId>appng-application-bom</artifactId>
                <type>pom</type>
                <scope>import</scope>
                <version>${appNGVersion}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

</project>
```

16.1. Using the appNGizer Maven plugin

The appNGizer Maven Plugin can be used to upload and install your application to several (local or remote) appNG instances.

By default, it binds to the `package` phase.

16.1.1. Goals

`upload`
This goal uploads the application to a local repository, but does not install it.

`install`
This goal uploads the application to a local repository and can also install it for a site. Optionally, this site can the be reloaded.

16.1.2. Configuration

General (default in braces):

- `endpoint` (`http://localhost:8080/appNGizer/`)
  the endpoint URL of appNGizer
- `sharedSecret` (*none*)
  the platform's shared secret to authenticate with

- `repository` (`Local`)
  the name of the **local** repository
- `baseauthUser` (*none*)
  the user name for basic authentication
- `baseauthPassword` (*none*)
  the password for basic authentication

`install` goal only:

- `activate` (`false`)
  if the installed archive should be activated for the site
- `site` (*none*)
  the name of the site to reload after installing the application

In a local environment, you usually do not set up cluster communication. In order to make reloading sites with the appNGizer maven plugin work, you have to set the site property `supportReloadFile` to `true`. The site reload will then be triggered by the existence of a marker file instead of a cluster event.

16.1.3. Example

```
<plugin>
    <groupId>org.appng.maven</groupId>
    <artifactId>appng-appngizer-maven-plugin</artifactId>
    <version>${appNGVersion}</version>
    <configuration>
        <endpoint>http://localhost:8080/appNGizer/</endpoint>
        <sharedSecret>TheSecret</sharedSecret>
        <repository>Local</repository>
        <activate>true</activate>
        <site>manager</site>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>install</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

16.2. Using the `local` profile

As an alternative deployment mechanism, you can use the `local` profile that builds an expanded version of your application directly into `${env.CATALINA_HOME}/webapps/ROOT/applications/`.

To make this work, these three preconditions must be fulfilled:

1. The environment variable `CATALINA_HOME` must be set correctly.
2. The application must have been installed through a local repository for at least one time.
3. The platform property `filebasedDeployment` must be set to `true`.

Note that changes in `application.xml` (`<roles>`, `<permissions>`, `<properties>`) do not get applied when using the `local` profile. You **must** deploy through a repository to apply these kind of changes.

You can even achieve to reload the site, as if using the appNGizer Maven Plugin (see above).

Therefore, set the site property `supportReloadFile` to `true` and let the `local` profile create the required marker-file. This can be done by using the `maven-antrun-plugin` with a `<touch>` target:

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <goals>
        <goal>run</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <target>
          <touch file="${env.CATALINA_HOME}/webapps/ROOT/repository/manager/.reload"
/> ①
        </target>
      </configuration>
    </execution>
  </executions>
</plugin>
```

① The path to the marker-file, where `manger` represents the name of the site to reload.

17. Using Camunda BPMN

A version of appNG is available that ships with the libraries for the Camunda BPMN workflow enigne: `appng-application-camunda-1.18.0-RC1.war`. When using this version of appNG, your application can easily leverage the power of Camunda, without worrying about bundling the right libraries. To use Camunda, add the following to the `pom.xml` of your application:

```
<dependencies>
    <dependency>①
        <groupId>org.appng</groupId>
        <artifactId>appng-camunda</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>②
            <groupId>org.appng</groupId>
            <artifactId>appng-application-bom</artifactId>
            <type>pom</type>
            <scope>import</scope>
            <version>${appNGVersion}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

① Add a dependency to `appng-camunda`.

② Use appNG's bill of materials (BOM).

Next, you have to define some Camunda specific beans in your `beans.xml`:

```
<bean id="transactionManager" class=
"org.springframework.jdbc.datasource.DataSourceTransactionManager">①
  <property name="dataSource" ref="datasource" /> ②
</bean>

<bean id="processEngineConfiguration" class=
"org.camunda.bpm.engine.spring.SpringProcessEngineConfiguration">
    <property name="processEngineName" value="engine" />
    <property name="dataSource" ref="datasource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jobExecutorActivate" value="false" />
    <property name="deploymentResources" value="classpath*:*.bpmn" />
</bean>

<bean id="processEngine" class="
org.camunda.bpm.engine.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>

<bean id="repositoryService" factory-bean="processEngine" factory-method=
"getRepositoryService" />
<bean id="runtimeService" factory-bean="processEngine" factory-method=
"getRuntimeService" />
<bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />
<bean id="historyService" factory-bean="processEngine" factory-method=
"getHistoryService" />
<bean id="managementService" factory-bean="processEngine" factory-method=
"getManagementService" />
<bean id="caseService" factory-bean="processEngine" factory-method="getCaseService" />
<bean id="formService" factory-bean="processEngine" factory-method="getFormService" />
```

① We need a `org.springframework.transaction.PlatformTransactionManager`. If your application uses JPA, use `JpaTransactionManager` here instead!

② Refers to the predefined `javax.sql.DataSource`, see section pre-defined beans.

Your application needs a database to make Camunda work, even if the application itself does not store anything in that database.

For more details about Spring and Camunda, check out the guide Get started with Camunda and the Spring Framework.

17.1. Implementing user tasks

BPM user tasks can easily be implemented using standard appNG actions and datasources.

You can use a org.appng.camunda.bpm.TaskWrapper to wrap a org.camunda.bpm.engine.task.Task.

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.appng.api.ActionProvider;
import org.appng.api.DataContainer;
import org.appng.api.DataProvider;
import org.appng.api.Environment;
import org.appng.api.FieldProcessor;
import org.appng.api.Options;
import org.appng.api.Request;
import org.appng.api.model.Application;
import org.appng.api.model.Site;
import org.appng.camunda.bpm.TaskWrapper;
import org.camunda.bpm.engine.FormService;
import org.camunda.bpm.engine.TaskService;
import org.camunda.bpm.engine.form.TaskFormData;
import org.camunda.bpm.engine.task.IdentityLink;
import org.camunda.bpm.engine.task.Task;
import org.camunda.bpm.engine.variable.VariableMap;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserTasks implements DataProvider, ActionProvider<TaskWrapper> {

    private TaskService taskService;
    private FormService formService;

    @Autowired①
    public UserTasks(TaskService taskService, FormService formService) {
        this.taskService = taskService;
        this.formService = formService;
    }

    public DataContainer getData(Site site, Application application, Environment
environment, Options options,
            Request request, FieldProcessor fp) {
        String taskId = options.getString("task", "id");

        Task task = taskService.createTaskQuery().taskId(taskId).initializeFormKeys()
.singleResult();
        List<IdentityLink> identityLinks = taskService.getIdentityLinksForTask(taskId
);
        VariableMap variables = taskService.getVariablesTyped(taskId);
        TaskWrapper taskWrapper = new TaskWrapper(task, identityLinks, variables);②

        TaskFormData taskFormData = formService.getTaskFormData(taskId);
        taskWrapper.addFormFields(fp, taskFormData, "Mandatory!");③

        DataContainer dataContainer = new DataContainer(fp);
```

```
        dataContainer.setItem(taskWrapper);
        return dataContainer;
    }

    public void perform(Site site, Application application, Environment environment,
Options options, Request request,
            TaskWrapper formBean, FieldProcessor fieldProcessor) {
        String taskId = options.getString("task", "id");
        Map<String, Object> variables = new HashMap<>();
        Map<String, Object> formFields = formBean.getFormFields();
        // process form fields ④
        variables.put("assignee", "John Doe");
        taskService.complete(taskId, variables);⑤
        fieldProcessor.addOkMessage("Task " + taskId + " completed!");
    }

}
```

① Use Camunda's `TaskService` and `FormService`

② Create a `TaskWrapper` from a `Task`, the `identityLinks` and the `VariableMap`

③ Dynamically add the form fields defined for this usertask to the `FieldProcessor`. See TaskWrapper#addFormFields for details.

④ Retrieve the values for the dynamically added form fields and process them.

⑤ Complete the task.

18. Appendix

18.1. List of icons

TODO

18.2. Link Modes