Framework Reference Documentation

Table of Contents

This part of the technical documentation will cover all of the core technologies used within the Framework.

Chapter 1. The Domain Entity

Having a quality understanding of how to configure *domain entities* is essential to build other types of configuration offered by the Framework. Furthermore, it is a great place to be considered as an entry point in using the Framework. This chapter will describe the primary concepts and usage behind the *domain entity*.

1.1. Introduction to the Domain Entity

At it's core, the *domain entity* is nothing more than a simple encapsulated Java object. It is the blueprint for the objects that the Framework will create and use, similarly to the way a class declaration is used to create an instance of an object in Java.

Aside from being a straightforward Java bean, the *domain entity* may have a specific Framework metadeta decorating it's fields (the Framework refers to these as Config Annotations) which will be used to instruct the Framework on how to handle the *domain entities* created from this *domain entity* during certain events.

See the following sample *domain entity*:

```
@Domain("person")
@Getter @Setter @ToString
public class Person {
    private String firstName;
    private String lastName;
}
```

In this example `Person` has been defined as a *simple domain entity* with getters and setters defined for each of it's fields.

It may be intuitive that this entity is a data-centric model created solely for the purpose of collecting data; it is. While this example is a simple one, it will be used in the examples to follow to demonstrate the domain modeling feature.

Domain Entity Config Annotations

In the previous example, `@Domain` was used to decorate a class declaration so that the Framework used to identify `Person` as a domain entity. Domain defines the root or topmost level of a *domain entity*. The Framework supports other types of *domain entity* markers as well.

The config annotations that can be used to register domain entities are:

- Domain
- Model

Defining Behavior

There are many different variations of how the Framework can utilize domain entities. One such manner is to define a `ListenerType` on the domain entity that provides the Framework with extra instruction on how to handle an instance of it during specific scenarios. Consider the following example:

```
@Domain(value = "person", includeListeners = { ListenerType.persistence })
@Repo(value = Database.rep_mongodb, cache = Cache.rep_device)
@Getter @Setter @ToString
public class Person {
    @Id
    private String id;
    private String firstName;
    private String lastName;
}
```

This example is very similar to the first, except the decorative annotation `@Domain` contains an additional attribute with a value of `ListenerType.persistence` and a `@Repo` config annotation. This configuration instructs the framework to have the underlying persistence layer perform certain actions on instances of this domain entity during the Framework lifecycle. Consider a use case where there is a need to update the values of a particular entity within the application's database according to what is stored in an instance of this `Person` object. Specifying `ListenerType.persistence` as a *listener* for this domain entity will handle such a scenario.

NOTE	There are other `ListenerType` types and other mechanisms to configure the domain entity behavior available on `Domain` and each of the other domain entity config annotations. See the specific config annotation for more details (e.g. Domain, Model).

1.2. Core versus View

A key concept to the Framework in terms on the *domain entity* is understanding the difference between a **core domain entity** and a **view domain entity**.

The Framework is reponsible for handling two primary sets of data in terms of configuration data:

- The "view" data which tells the framework how to render the graphical user interface.
- The "core" data that the application creates, reads, updates, and deletes.

Therefore it is essential that it knows how to differentiate between what is *core* and what is *view*. At the same time to take full advtange of abstraction capabilities within the Framework, the domain entities are essentially created in the same fasion. How the Framework will interpret between the domain entities is entirely up to how they are configured given the configuration metadata, which will be explained in later sections of this chapter (see Mapping Params and Annotation Config).

Definition:

A **core domain entity** is a *domain entity* who's primary responsibility is maintaining the integrity of the data contained within the application.

Definition:

A **view domain entity** is a *domain entity* who's primary responsbility is maintaining instructions on how the graphical user interface be rendered.

1.3. Defintion Structure

1.3.1. Simple Domain Entities

The `Person` *domain entity* previously defined is a representation of what is understood to be a *simple domain entity*. A *simple domain entity* is one who's field data does not contain other *domain entity* types. In other words, it's defined fields consist only of simple types.

While not always the case, simple domain entities are typically associated with _core domain entities.

1.3.2. Complex Domain Entities

The Framework supports the notion of "nesting" *domain entities* within each other.

Consider the following example of a *view domain entity* representing a form for the aforementioned `Person`:

Person.java

```
@Model
@Getter @Setter @ToString
public class VFPersonForm {

    @TextBox
    private String firstName;

    @TextBox
    private String lastName;

    private Address address;
}
```

Address.java

```
@Model
@Getter @Setter @ToString
public class Address {

    @TextBox
    private String line1;
    ...
}
```

In this example, both `VFPersonForm` and `Address` are decorated with `@Model`. Therefore, `VFPersonForm` is a *complex domain entity* because it contains an instance of `Address`.

This scenario contains only one level of nesting between `VFPersonForm` and `Address`. It is not uncommon to have multiple layers of "nesting" within a *domain entity*, hence giving rise to the term *complex*. Consider a common scenario in HTML where a page contains a section, which contains a tab, which contains a component, which contains a... the context of what could be rendered in the view could be boundless. The Framework is able to support this via "nesting" and using these *complex domain entities*.

While not always the case, complex domain entities are typically associated with *view domain entities*.

1.3.3. Defining a View

The UI Framework expects a specific pattern in terms of where components are placed. Each of the View Config Annotations has explicitely defined which components are to be placed where.

The same is true for setting up the standard page view. The following configuration is a typical view configuration to get started:

Sample View Entity 1

```
@Domain(value="sample_view1", includeListeners = { ListenerType.websocket })
@Repo(Database.rep_none)
public class VRSampleView1 {

    @Page
    private VPMain vpMain;

    @Model @Getter @Setter
    public static class VPMain {

        @Tile
        private VTMain vtMain;
    }

    @Model @Getter @Setter
    public static class VTMain {

        @Section
        private VSMain vsMain;
    }

    @Model @Getter @Setter
    public static class VSMain {

        // Components here...
    }
}
```

As mentioned, this is the standard view config which can be expanded upon for an application's specific needs. The primary point to note is the structure of **Domain Entity** > **Page** > **Tile** > **Section**. Only once this structure has been defined will. For more specific information on what components can go where, please review the View Config Annotations section.

1.4. Loading the Domain Entities

The `com.antheminc.oss.nimbus.domain.config.builder` package is reponsible for consuming the *domain entity* information during the initial loading process. On application startup, the Framework consumes the defined *domain entity* configurations and stores that information included within them in memory.

The registration process for domain entities starts with a registered Spring bean of type `DomainConfigBuilder`. This instance of `DomainConfigBuilder` uses the Spring Framework's scanning capabilities to retrieve `Domain` configuration defined within specific packages.

Configuring Base Packages

The packages that are scanned in this manner can and should be defined via the application properties using the standard Spring `PropertyPlaceholderConfigurer` mechanism (using *application.properties*, *application.yml*, etc.).

The property used to identify packages for registration is: `domain.model.basePackages`.

```
domain:
  model:
    basePackages:
      - com.antheminc.oss.nimbus.entity
      - com.acme.app.model.feature_a.core
      - com.acme.app.model.feature_a.view
      - com.acme.app.model.feature_b.core
      - com.acme.app.model.feature_b.view
```

NOTE The order of which the entities are loaded may play an impact on certain framework features. It is recommended to always load *core* domain config prior to *view* domain config.

1.5. The Param Interface

Consider that the Framework is acting much like an object oriented programming language in that a *domain entity* definition is playing the part of a class declaration. The next step in the Framework lifecycle is to use that definition to create an instance, much like the way an object is instantiated in an Objected Oriented Programming language.

This section focuses on the creation and use of the resulting object created by using the configuration consumed from the *domain entity*. The framework refers to this as the *param*, created using the *Param* interface.

This chapter will focus on explaining the *Param* interface in depth.

1.5.1. Introduction

After the framework registers domain entity definitions during application startup, it is creating what the Framework understands as a `Param`. The `Param` interface defines a number of properties on it that both the framework and UI framework will use to perform a given operation. Some examples might be, setting the value of the object, setting the object's visibility, or setting whether or not the object should have validation applied to it (if it is a form element). Whatever the case, the important thing to note is that the `Param` interface is collectively the "model" data that is used everywhere throughout the Framework.

While understanding the core concepts of the `Param` is not essential for building out the UI of the application, it is important to understand it when dealing with certain functional scenarios. For example, an application can be created using the Framework by simply using the View Config Annotations and Conditional Config Annotations; however, when needing to manipulate the model directly (perhaps in a rule or SpEL expression), having an understanding of `Param` will be extremely useful.

1.5.2. Pathing

As previously explained, every Java variable created under a domain entity definition will eventually result in the creation of a *param*. Each param that is created in this way has a unique **path** by which it can be identified through using the framework's Command Query DSL.

Introduction

The path is represented as a URI address that uniquely identifies a param's location within the Framework. It is built by traversing through a Java class's members, with a `/` denoting a child member (child members are also often referred to as "nested" members).

An example of some param paths can be found in the sample code below:

```
@Domain("petshop")
@Getter @Setter
public class PetShop {

    // Param Path: /petshop:1/name
    private String name;

    // Param Path: /petshop:1/address
    private Address address;

    // Param Path: /petshop:1/boardedPets
    private List<Pet> boardedPets;

    // Collection elements are also retrievable by supplying an index
    // immediately following member name.
    // Collection Element Param Path with index 0:
    //   /petshop:1/boardedPets/0
    // Collection Element Param Path with index 1:
    //   /petshop:1/boardedPets/1
    // Collection Element Param Path with index n (where n is an
    // integer >= 0):
    //   /petshop:1/boardedPets/n

    @Model
    @Getter @Setter
    public static class Address {

        // /petshop:1/address/street
        private String street;

        ...
    }

    @Model
    @Getter @Setter
    public static class Pet {

        // /petshop:1/boardedPets/0/name
        // /petshop:1/boardedPets/1/name
        // /petshop:1/boardedPets/.../name
        private String name;

        ...
    }
}
```

Please note that in the example above all of the provided param paths are of the form: `/petshop:1/⋯`. The index `1` is known as a `refId`, which will be explained in the following section.

RefId

A *refId* is a unique identifier used to identify a particular instance of *domain entity* that has been created.

Path Variable Resolver

The Command Query DSL offers certain reserved variables or keywords that can be used in a path. The Framework uses a default implementation of `CommandPathVariableResolver` to resolve a param path from a provided [param-pathing-context].

Param Context

NOTE	This section is a work in progress.

Property Placeholders

The default implementation of `CommandPathVariableResolver` will attempt to first resolve `${…}` placeholders in the given path, replacing them with corresponding property values. Unresolvable placeholders with no default value is not supported.

Reserved Keywords

After resolving property placeholders reserved framework keywords will be resolved. The framework provides support for the following reserved keywords:

Keyword	Description
`/p`	Denotes the start of the param path. This is only needed to be given when using a full URL or switching param contexts.
`/.m`	Resolves to the path of the param that is mapped to the param identified by the preceeding path. (e.g. `PARAM_PATH/.m` returns the mapped param of the param identified by `PARAM_PATH`.
`/.d`	Resolves to the path of the param that is the root domain of the param identified by the preceeding path. (e.g. `PARAM_PATH/.d` returns the domain param that owns the param identified by `PARAM_PATH`.
`<!#elemId!>`	
`<!#env!>`	
`<!#refId!>`	
`<!#self!>`	
`<!#this!>`	Resolves to the path of the param in the current context.

1.5.3. Param Subclasses

There are many different types of `Param` implementations that are available. Each describes the model as created in the domain entity in such a way that the Framework is able to understand. For specific details about the implementations, see the `com.antheminc.oss.nimbus.domain.model.state` package.

SampleEntity.java

```
@Domain("sample_entity")
@MapsTo.Type(SampleCoreEntity.class)
public class SampleEntity {

    // Creates a leaf param
    private String leaf;

    // Creates a nested param
    private NestedEntity nestedEntity;

    @Model @Getter @Setter
    public static class NestedEntity {

        // Creates a leaf param inside of a nested param
        private String nestedLeafEntity;
    }

    // Creates a collection param, and potentially collection element params
    private List<NestedEntity> collectionEntity;

    // Creates a leaf param and a mapped param
    @MapsTo.Path
    private String mappedEntity;
}
```

Refer to `SampleEntity.java` above when reviewing the different classifications of `Param` below.

Leaf Param

A leaf param is a `Param` instance who has no children or nested params. A leaf param is represented by the interface `LeafParam`.

The variable `leaf` in the `SampleEntity.java` domain entity would be constructed by the Framework as a `LeafParam`.

Nested Param

A nested param is a `Param` instance who has children params. A nested param is represented by the interface `Model`.

The variable `nestedEntity` in the `SampleEntity.java` domain entity would be constructed by the Framework as a `Model`.

Collection Param

A collection param is a `Param` instance who is defined as an array or collection type. A collection param is represented by the interface `ListParam`.

The variable `collectionEntity` in the `SampleEntity.java` domain entity would be constructed by the

Framework as a `ListParam`.

Collection Element Param

A collection element param is a `Param` instance contained as a child of a `ListParam`. It represents a child of a collection in the same sense that Java considers a collection element to belong to a `java.util.List`. A collection param element is represented by the interface `ListElemParam`.

The variable `collectionEntity` in the `SampleEntity.java` domain entity would contain 0 or more `ListElemParam` instances.

Mapped Param

A mapped param is a `Param` instance who's entity definition has been annotated with `@MapsTo.Type`. A mapped param is represented by the interface `MappedParam`.

The variable `mappedEntity` in the `SampleEntity.java` domain entity would be constructed by the Framework as a `MappedParam`.

For more information about mapped params and their behaviors, see Mapping Params.

1.5.4. State

The methods `getState()` and `setState(…)` of `Param` are available for retrieving and setting the stored value of a param located at a particular path.

For example, consider working with a param at the domain level using the following Domain Entity definition:

Sample Domain Entity POJO

```
@Domain("sample_object")
public class SampleObject {

    private String a;
    private NestedObject1 b;

    @Model
    public static class NestedObject1 {

        private String c;
        private NestedObject2 d;
    }

    @Model
    public static class NestedObject2 {

        private String e;
    }
}
```

Consider also that one domain entity of `SampleObject` has been created with *refId* = 1, and has been set with the following data:

Set State Sample JSON

```
{
    "a": "1",
    "b": {
        "c": "2",
        "d": {
            "e": "3"
        }
    }
}
```

This means that the framework will have created params for the `sample_object` domain entity with *refId* = 1, and the user will be able to retrieve that data using the framework's DSL. The framework DSL to retrieve this item might look like the following:

```
/p/sample_object:1/_get
```

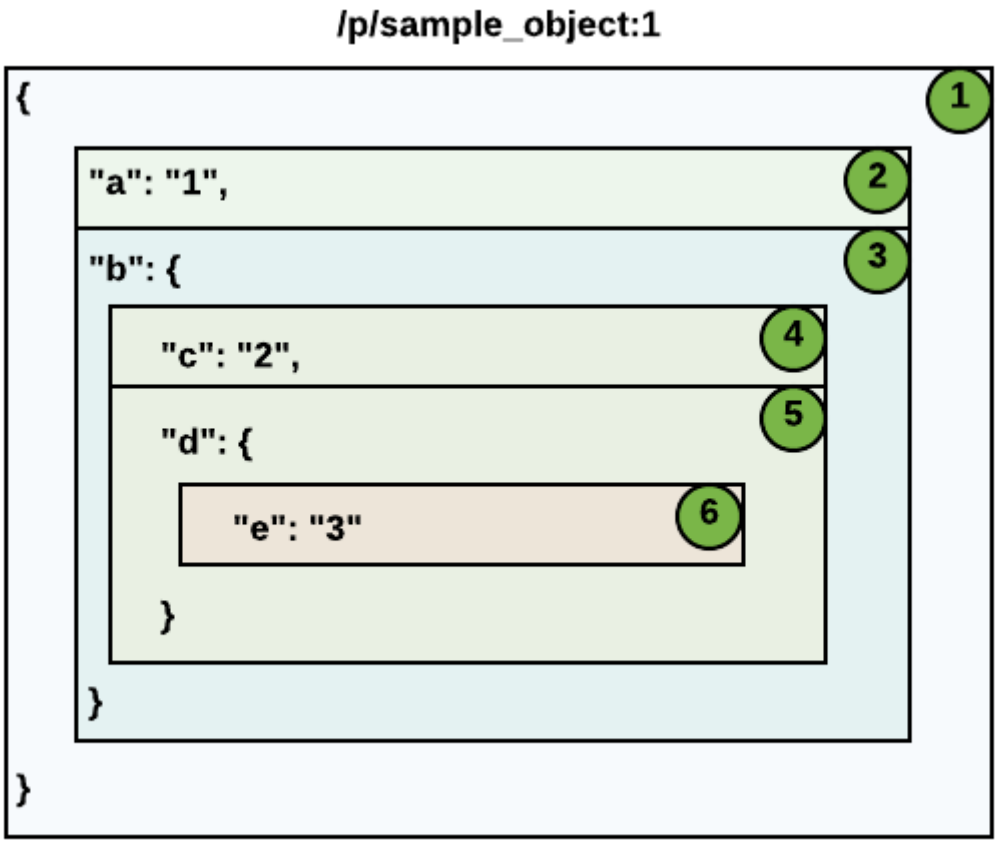
The framework retrieves state by recursively iterating over a param's child params (also commonly referred to as nested params). If thinking of `/p/sample_object:1` as the root entity, then using the domain entity definition for `SampleObject`, the children entities would look like the following:

- /p/sample_object:1
- /p/sample_object:1/a
- /p/sample_object:1/b
- /p/sample_object:1/b/c
- /p/sample_object:1/b/d
- /p/sample_object:1/b/d/e

Efficiency

For the sake of effiency, it is important to be aware at a high level of how the framework is handling operations regarding state. When working with simple domain entities, such as an individual String or Integer the operation is straightforward. When working at a layer which contains nested paths, caution should be exercised to understand if the get/set state is truly needed at that layer to avoid invoking unnecessary operations.

For the previous example, when `setState(⋯)` is invoked on root domain entity using *refId*=1, the framework is actually invoking more than a single `setState(⋯)`. See the diagram below:



This diagram attempts to render a visual representation of how each of the `setState(…)` invocations would occur when setting the state of a particular object in JSON. It illustrates that if a user is attempting to execute a `setState(…)` call on the object located at `/p/sample_object:1`, 6 `setState(…)` invocations will occur (See 1-6 in the diagram above).

Consider a scenario where only a particular value within a domain entity is needed to be set. In that case, we can target the unique param for that value by using it's path. (e.g. update the param at `/p/sample_object:1/b/d/e` instead of the entire param at `/p/sample_object:1`. See #6 in the diagram above.) This would mean that the framework has avoided the processing of 5 extra `setState(…)` calls, when it only needed to execute 1.

Chapter 2. Configuration

The core concept of the Framework is that one is able to write the different types of Configuration to build an application. *Configuration* is a high-level term used to describe the code that ultimately drives the logic of the Framework. The topmost and most common type of configuration will typically be found within an application's *domain entities*.

This chapter will explore and explain the numerous configuration features of the Framework.

2.1. Annotation Config

The Framework offers several different means of configuration the application through Java based annotations. As seen in several of the previous examples, much of the emphasis of configuration is on decorating domain entity variables.

The most common types of annotation configs used in an application fall under the following categories:

- View Config Annotations - Instruct the UI to render a specific view component (e.g. textbox, calendar, accordion)
- Core Config Annotations - Execute functional instructions or commands within the Framework
- Conditional Config Annotations - Execute common functional instructions based on boolean SpEL expressions

2.2. Mapping Params

The section Core versus View explains the difference between a **core domain entity** and a **view domain entity**. While a **core domain entity** is the primary holder of the data, the **view domain entity** is primarily concerned with the display of the data.

To establish a connection between the core and the view (similar to how an MVC application does), the Framework introduces the concept of **mapping** the state between `Param` instances. This provides for a two-way binding between the state of mapped core and view params in that when the state of the core is updated, the state of the view will also be updated. The same is true when updating the state of a mapped view param, in that the state of the core param would be updated.

2.2.1. Configuring a Mapping

The process for defining a mapping between params is relatively straightforward. See the following example:

Sample Core Entity

```
@Domain("sample_core", includeListeners = { ListenerType.persistence })
@Repo(value = Database.rep_mongodb, cache = Cache.rep_device)
@Getter @Setter @ToString
public class SampleCoreEntity {

    private String name;
}
```

This class represents the definition of our **core domain entity**. Params created from this defintion will be mapped to from view Params.

Sample View Entity 1

```
@Domain(value="sample_view1", includeListeners = { ListenerType.websocket })
@Repo(Database.rep_none)
@MapsTo.Type(SampleCoreEntity.class)
public class VRSampleView1 {

    // Page/Tile/Section/Form declarations omitted for brevity.

    @Model @Getter @Setter
    @MapsTo.Type(SampleCoreEntity.class)
    public static class SampleForm {

        @TextBox
        @MapsTo.Path
        @Label("Enter name: ")
        private String name;
    }
}
```

This class represents the definition of a **view domain entity**. Notice that both `VRSampleView1` and `SampleForm` map to `SampleCoreEntity.class`. Now that `name` is decorated with `@MapsTo.Path`, the Framework knows that the textbox component defined at `SampleForm.name` is mapped to `SampleCoreEntity.name`. Hence the value displayed in the textbox would be equal to the state found in `SampleCoreEntity.name`.

Sample View Entity 2

```
@Domain(value="sample_view2", includeListeners = { ListenerType.websocket })
@MapsTo.Type(SampleCoreEntity.class)
@Repo(Database.rep_none)
public class VRSampleView2 {

    // Page/Tile/Section/Form declarations omitted for brevity.

    @Model @Getter @Setter
    @MapsTo.Type(SampleCoreEntity.class)
    public static class SampleForm {

        @FieldValue
        @MapsTo.Path("name")
        @Label("Hello: ")
        private String personName;
    }
}
```

This class represents another definition of a **view domain entity**. In this scenario, note that `SampleForm.personName` is decorated with `@MapsTo.Path("name")`, the Framework knows that the field value component defined at `SampleForm.personName` is mapped to `SampleCoreEntity.name`. Hence the value displayed in the field value component would be equal to the state found in `SampleCoreEntity.name`.

Both of these scenarios illustrates the ability to create one or more views for a core domain entity, which is a very typical scenario when displaying data in any application. This allows users of the Framework to re-use core domain data using a multitude of views, using whichever components are necessary to achieve specific funtionality.

For more information on the `@MapsTo.Path` and `@MapsTo.Type` annotations, see the MapsTo section.

2.3. The Rules Engine

TODO

2.4. Business Process Model and Notation (BPMN)

TODO

2.5. Change Log

Any command execution now gets captured in a collection called *changelog* in mongodb.

Table 1. ChangeLog Attributes

Attribute	Type	Description
on	Date	command config execution date & time
by	String	command config execution by userId
root	String	root entity on which command config is executed
refId	Long	root entity id on which command config is executed
action	Action	action performed during command config execution
url	String	command config complete uri
path	String	path of the param path that is being updated(_replace/_update) as part of command config execution
value	Object	value of the param that is being updated (_replace/_update) as part of command config execution

2.5.1. Usage Examples

Lets take the example domain named Test with an alias "testdsl". it has a form with a button marked with @Configs.

Test.java

```
@Getter @Setter
@Domain(value = "testdsl")
@Repo(Database.rep_mongodb)
@ToString
public class Test extends IdString{

    @Ignore
    private static final long serialVersionUID = 1L;

    private String status;

    @Form
    private Form form;

    @Model
    @Getter @Setter
    public static class Form {

        @Button
        @Configs ( {
            @Config(url="/p/testdsl2/_new"),
            @Config(url="/status/_update?rawPayload=\"Test_Status\"")
        })
        private String button;

    }
}
```

In above example, when the user (logged user 'testUser') triggers an action by clicking submit button (testdsl has a refId of 1), below command urls would get invoked in sequence:

1. ../p/testdsl:1/form/button/_get?b=$execute
2. ../p/testdsl2/_new
3. ../p/testdsl:1/status/_update?rawPayalod=\"Test_Status\"

changelog collection would now contain entries for all 3 command executions:

Table 2. ChangeLog collection entries

on	by	root	refId	action	url	path	value
current datetime	testUser	testdsl	1	_get	/p/testdsl/form/button/_get?b=$execute		
current datetime	testUser	testdsl2	2	_new	/p/testdsl2/_new		
current datetime	testUser	testdsl	1	_update	/p/testdsl/status/_update?rawPayload=\"Test_Status\"		

Chapter 3. The Command Query DSL

The *Command* is the instruction that the Framework understands to execute and come back with an output. It is similar to writing the traditional method calls for an action (such as a button click) to perform business logic, but attempts to introduce a standardized process via the use of a domain specific language (DSL) that the Framework can interpret. This is called the Command Query DSL.

The Command Query DSL is represented in URL format (commonly referred to as the *Command URL*) and is based on Query DSL.

Apart from the *host* (typical "host", or provider, of the application utilizing the Framework), the DSL's Command URL can be broken down into four main subsections of:

- **Client-alias** - *identification path used to segregate data/logic for one or multiple "clients". Supports clientCode, leaf-org-id, and appAlias*
- **Domain-alias** - *identification path within the framework to locate a defined object, or domain entity*
- **Action** - *instruction used to identify the operation to perform on the object identified by domain-root*
- **Query Parameters** - *A standard set of query parameters*

The following diagram shows each of the subsections of the Command Query DSL mentioned above in various examples in an "exploded" view:

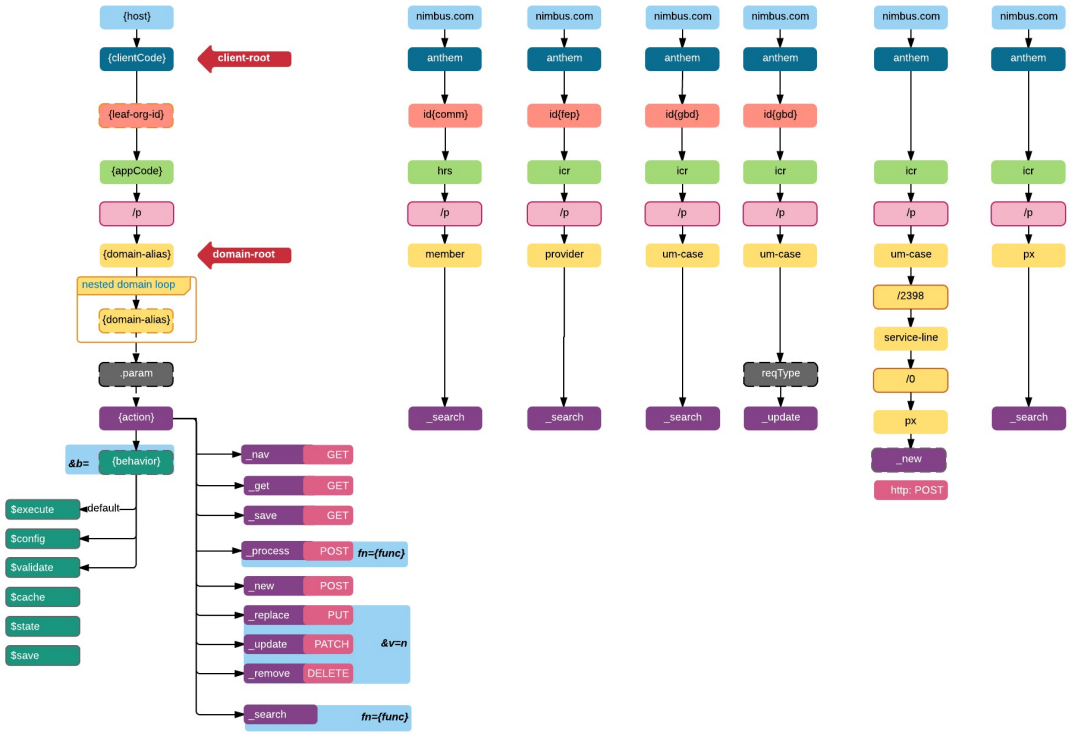


Figure 1. Command DSL Pattern Diagram

In the diagram above, several examples of Command URL's can be seen. By splitting the URL into it's individual parts, the diagram represents each of the Command URL's as a vertical section.

Requests sent to the Framework using each of these URLs perform different logic as defined by the Framework. The sections that follow will explain the logic defined for each of the individual subections of the Command URL as well as expain the Command Query DSL in-depth.

3.1. Client Alias

TODO

3.2. Domain Alias

The *domain alias* (also referred to as *path*) is the essential and active portion of working with the Command Query DSL. It is what is used to identify which object, or *domain entity*, the framework should perform a defined action on.

Each *domain alias* starts with a *domain root*, which is the highest level in the domain path used for identification purposes. This will correspond with values used in `@Domain`. From there, the Command Query DSL traverses the Java objects that are domain entities and finds a particular field identified by the remaining path. See The Param Interface for more information.

Using this path in the Command Query DSL can then be used to target a specific Action to perform, which will be discussed in the next section.

3.3. Actions

The *Action* is the third primary subsection of the Command URL. *Action* identifies which type of logic or instructions which should be applied to a given *domain entity*. The *domain entity* on which the action will be applied is identified by the *domain-alias* path given in the Command URL preceding the *action* parameter (see Domain Alias).

A Command URL without an *action* is not understood and hence not supported by the Framework. Hence, *action* is required.

See the following example:

```
@Config(url = "/domainobject/page/tile/section/form/_get?b=$execute")
```

In the example above, the action is **_get**. This states that we should perform the instructions defined for **_get** on the param located at `domainobject/page/tilesection/form`.

Each of the sections that follow will describe the logic defined within the Framework for that specific *action*.

3.3.1. New

_new: Creates a new instance for the model

3.3.2. Get

_get: Fetches the instance of the model referenced by the Id

3.3.3. Save

_save: Saves the model into the database

3.3.4. Replace

The action *_replace* sets the state of the param identified by *domain alias* to a new state, provided as a query parameter: *rawPayload*. When using the replace action, not providing explicit values for fields will result in any existing state within the identified param to be set to `null`. In a situation where only some values of the state need to be set and the others preserved, consider using Update.

Examples

The following class will be used in the *_replace* scenarios to follow.

```
@Domain("sample_entity")
@Getter @Setter
public class SampleEntity {

    private String single_element;
    private List<String> collection;
}
```

The following scenarios will also assume that the param at *sample_entity:1* has a pre-existing state of:

```
{
    "single_element": "empty_value",
    "collection": ["empty_value_1"]
}
```

Replacing a single element state

```
@Config(url = "sample_entity:1/single_element/_replace?rawPayload='hello'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "hello",
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be replaced with the `rawPayload`. The value for the field of `collection` is unaffected and retains it's previous value.

Replacing a collection element state

```
@Config(url = "sample_entity:1/collection/0/_replace?rawPayload='hello'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "empty_value",
    "collection": [ "hello" ]
}
```

As the path in this scenario is pointing to `sample_entity:1/collection/0`, only the collection element having index `0` for the field `collection` is subject to be replaced with the `rawPayload`. The value for the field of `single_element` is unaffected and retains it's previous value.

Replacing a collection state

```
@Config(url = "sample_entity:1/collection/_replace?rawPayload='[\"1\", \"2\"]'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "empty_value",
    "collection": [ "1", "2" ]
}
```

As the path in this scenario is pointing to `sample_entity:1/collection`, only the field `collection` is subject to be replaced with the `rawPayload`. The value for the field of `single_element` is unaffected and retains it's previous value.

Replacing a complex object

```
@Config(url = "sample_entity:1/_replace?rawPayload='{\"single_element\": \"hello\"}'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "hello"
}
```

As the path in this scenario is pointing to `sample_entity:1`, both fields are subject to be replaced with the `rawPayload`. In this scenario, only `single_element` has been provided which results in

`collection` having a `null` value.

NOTE	The important thing to note in this scenario, is that like all other scenarios the existing state is replaced. This means that if *collection* of the entity located at *sample_entity:1* had an existing value, it will be set to `null` since the *rawPayload* parameter did not explicitely set it.

Replacing an entity state with the value of another entity state

```
@Config(url = "sample_entity:1/single_element/_replace?rawPayload=<!json(PATH_TO_PARAM)!>)
```

where `PATH_TO_PARAM` is path which points to another param, relative to the location of `single_element`.

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be replaced with the `rawPayload`. The value to be set will be the same as the state of the object identified by `PATH_TO_PARAM`.

TIP	This scenario can be used to copy a param's state from one param to another.

Replacing an entity state with post data

```
@Config(url = "sample_entity:1/single_element/_replace")
```

Post Data

"hello"

Resulting State of sample_entity:1

```
{
    "single_element": "hello",
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be replaced with the HTTP post data. The value for the field of `collection` is unaffected and retains it's previous value.

Replacing an entity state to null

```
@Config(url = "sample_entity:1/single_element/_replace)
```

- Assumes post data is not present

Resulting State of sample_entity:1

```
{
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be replaced with the HTTP post data. In this case the post data is `null` so the resulting value for `single_element` will then be `null`. The value for the field of `collection` is unaffected and retains it's previous value.

3.3.5. Update

The action *_update sets individual fields of the state of the param identified by _domain alias* to the fields within the object provided as a query parameter: *rawPayload*. When using the update action, not providing explicit values for fields will result in any existing state within the identified param to be preserved. In a situation where only the state needs to be explicitly set, consider using Replace.

Examples

The following class will be used in the *_update* scenarios to follow.

```
@Domain("sample_entity")
@Getter @Setter
public class SampleEntity {

    private String single_element;
    private List<String> collection;
}
```

The following scenarios will also assume that the param at *sample_entity:1* has a pre-existing state of:

```
{
    "single_element": "empty_value",
    "collection": ["empty_value_1"]
}
```

Updating a single element state

```
@Config(url = "sample_entity:1/single_element/_update?rawPayload='hello'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "hello",
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be updated with the value of `rawPayload`. The value for the field of `collection` is unaffected and retains it's previous value.

Updating a collection element state

```
@Config(url = "sample_entity:1/collection/0/_update?rawPayload='hello'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "empty_value",
    "collection": ["hello"]
}
```

As the path in this scenario is pointing to `sample_entity:1/collection/0`, only the collection element at index `0` for the field `collection` is subject to be updated with the value of `rawPayload`. The value for the field of `single_element` is unaffected and retains it's previous value.

Updating a collection state

```
@Config(url = "sample_entity:1/collection/_update?rawPayload='\"new_value\"'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "empty_value",
    "collection": ["empty_value_1", "new_value"]
}
```

As the path in this scenario is pointing to `sample_entity:1/collection`, only the field `collection` is subject to be updated with the value of `rawPayload`. Performing an *_update* call on a collection essentially performs an **add** operation. The value for the field of `single_element` is unaffected and retains it's previous value.

TIP	If needing to replace elements within the collection as a whole, consider using Replace.

It is also permissible to pass an array as a payload.

```
@Config(url = "sample_entity:1/collection/_update?rawPayload='[\"new_value_1\", \"new_value_2\"]'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "empty_value",
    "collection": ["empty_value_1", "new_value_1", "new_value_2"]
}
```

In this scenario, **all of the array elements** within `rawPayload` would be added to the collection.

Updating a complex object

```
@Config(url = "sample_entity:1/_update?rawPayload='{\"single_element\": \"hello\"}'")
```

Resulting State of sample_entity:1

```
{
    "single_element": "hello",
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1`, both fields are subject to be updated with the value of `rawPayload`. Performing an *_update* call in this call preserves any previously existing state for fields not specified in the `rawPayload`. Consequently, the value for the field of `single_element` is updated, while `collection` is unaffected and retains it's previous value.

TIP	Nested domain models can become quite large! Use *_update* when needing to target specific values within a domain entity.

Updating an entity state with the value of another entity state

```
@Config(url = "sample_entity:1/single_element/_update?rawPayload=<!json(PATH_TO_PARAM)!>)
```

where `PATH_TO_PARAM` is path which points to another param, relative to the location of `single_element`.

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be updated with the value of `rawPayload`. The value to be set will be the same as the state of the object identified by `PATH_TO_PARAM`.

TIP	This scenario can be used to copy a param's state from one param to another.

Updating an entity state with post data

```
@Config(url = "sample_entity:1/single_element/_update")
```

Post Data

"hello"

Resulting State of sample_entity:1

```
{
    "single_element": "hello",
    "collection": ["empty_value_1"]
}
```

As the path in this scenario is pointing to `sample_entity:1/single_element`, only the field `single_element` is subject to be updated with the HTTP post data. The value for the field of `collection` is unaffected and retains it's previous value.

3.3.6. Delete

_delete: Removes the model from the database

3.3.7. Search

_search: Searches the model based on a search criteria

3.3.8. Process

_process: Executes asigned workflow process or custom hanlders

3.3.9. Nav

_nav: navigates to the model

3.4. Query Params

TODO

Chapter 4. Appendix

4.1. Config Annotations

This section contains instructional reference material for each of the config annotations available within the Framework. Use this section when needing to identify specific features of a config annotation.

For more information about how config annotations are applied within the framework, see Annotation Config.

4.1.1. Conditional Config Annotations

This section covers the *conditional behavior* that can be applied to params. In this context, *conditional behavior* refers to performing a set of actions based on a given condition. The condition is evaluated by using the powerful capabilities of *SpEL (Spring Expression Language)*.

Typically this *conditional behavior* is evaluated from the context of the *conditionally decorated param* (the param that is annotated with a conditional annotation), meaning that the condition will inspect the *decorated param* object to infer if the condition should be `true` or `false`.

The following example shows what a conditional configuration might look like:

```
@SampleConditional(when = "state == 'YES'")
@Radio(postEventOnChange = true)
@Values(YesNo.class)
private String question;
```

The `when` condition is effectively stating, when `question.getState().equals("YES")`. In this case, since `question` is the *decorated param*, we are evaluating from the context of `question`. This is why `state` is understood in the *SpEL* condition `state == 'YES'`.

NOTE	In this scenario `@SampleConditional` is not a real annotation understood by the framework, but it illustrates the idea that when this particular *condition* is `true`, we should perform some sort of behavior. We defer that logic to *State Event Handlers*.

AccessConditional

`@AccessConditional` allows to control the access of the specific area/functionality within the application based on User's assigned role(s) and permission(s).

Example:

```
@AccessConditional(whenAuthorities="?[#this == 'entity_assign'].empty",
p=Permission.HIDDEN)
private Section_EntityAssignment vsEntityAssignment;
```

Above configuration would hide the entity assignment section for user(s) who do not have access to "entity_assign" action.

ActivateConditional

`@ActivateConditional` is an extension capability provided by the framework. The annotation is used to conditionally activate/deactivate the param based on a SpEL condition. This annotation can be triggered for multiple events. Framework provides default event handling for this annotation on StateChange and StateLoad.

ActivateConditional.java

```
@CheckBox(postEventOnChange=true)
@ActivateConditional(when="state != 'true'", targetPath="/../sectionG_skip")
private String skipSectionG;

private sectionG_Skip sectionG_skip;

@Radio(postEventOnChange=true)
@Model.Param.Values(value=YNType.class)
@ActivateConditional(when="state == 'No'", targetPath="/../q3Level2")
private String answerYesOrNo;

private Q3 q3Level2;

@CheckBoxGroup(postEventOnChange=true)
@Model.Param.Values(value=Days.class)
@ActivateConditional(when="state == new
String[]{'Sunday'}",targetPath="/../../../sundayDeliverySection")
private String[] deliveryDays;

/*Check if selection contains one or more specific values*/
@CheckBoxGroup(postEventOnChange=true)
@Model.Param.Values(value=Days.class)
@ActivateConditional(when="state != null && state.?[#this=='Sunday'] != new
String[]{}",targetPath="/../../../sundayDeliverySection")
private String[] deliveryDays;

@CheckBoxGroup(postEventOnChange=true)
@Model.Param.Values(value=VisitCount.class)
@ActivateConditional(when="state != null && state.length >
2",targetPath="/../../section")
private String[] visits;
```

ConfigConditional

`@ConfigConditional` is an extension capability provided by the framework. The annotation is used to conditionally execute `@Config` calls based on a *SpEL* based condition. This annotation can be triggered for multiple events. Framework provides default event handling for this annotation on StateChange.

ConfigConditional.java

```
@ConfigConditional(
    when = "state == 'Completed'", config = {
    @Config(url="<!#this!>/../state/_update?rawpayload=\"Closed\""),
    @Config(url="/p/dashboard/_get")
})
private String status;
```

NOTE	In the above example , whenever there is statechange of status and the status is changed to **Completed**, the Configs will be executed.

EnableConditional

`@EnableConditional` is an extension capability provided by the framework. The annotation is used to conditionally activate/deactivate the param based on a SpEL condition. This annotation can be triggered for multiple events. Framework provides default event handling for this annotation on StateChange and StateLoad. The difference between this annotation and `@ActivateConditional` is this annotation only affects the "enabled" state whereas `@ActivateConditional` affects both "enabled" and "visible" state.

EnableConditional.java

```
@EnableConditional(when="state == 'hooli'", targetPath="../enable_p2")
private String enable_p1;

private String enable_p2;
```

ExpressionConditional

`@ExpressionConditional` is a very versatile conditional annotation that can be used to apply any logic that is able to be crafted using *SpEL* expressions.

The framework provides default event handling for this annotation during the events:

- OnStateLoad
- OnStateChange

```
@ExpressionConditional(when="null == state && onLoad()",
then="setState(T(java.time.LocalDate).now())")
private LocalDate initDateOnLoad;
```

CAUTION	The observant will notice that `@ExpressionConditional` is versatile in that it could be used to perform the same logic achieved by several other framework defined *conditional annotations*. Ideally it should not be used when another *conditional annotation* is available to achieve the same behavor.

ValidateConditional

`@ValidateConditional` is used to conditionally set validations that should appear for a param based on a *SpEL* condition. This annotation can be triggered for multiple conditions, if necessary.

The framework provides default event handling for this annotation on StateChange.

Configuring Conditional by Group

`@ValidateConditional` works by first evaluating the `when` attribute by means of a *SpEL* condition. When the `when` condition is `true`, the framework will attempt to identify a subset of params with Validation constraints and apply validation logic to those params. See the following example:

ValidateConditional Sample 1

```
@ValidateConditional(when = "state == 'dog'", targetGroup = GROUP_1.class)
@TextBox(postEventOnChange = true)
private String petType;

@NotNull
@TextBox
private String petName;

@NotNull(groups = { GROUP_1.class })
@TextBox
private String dogFood;

@NotNull(groups = { GROUP_2.class })
@TextBox
private String catFood;
```

ValidateConditional Sample 1 - Results

In the above example, assuming the state of `petType` is `"dog"`, the validations that will be applied are: `petName` and `dogFood`.

- `petName` - There are no groups associated with `petName`, hence it is seen as a *static validation* and always applied.
- `dogFood` - `GROUP_1.class` is present within the `@NotNull's` `` `groups `` attribute, and this is matching `petType's` `` `@ValidateConditional's `` `targetGroup` attribute(`GROUP_1.class`) and is applied.

The validations that will not be applied are: `catFood`.

- *catFood* - While *catFood* has an entry in `@NotNull's` `` `groups `` attribute(`GROUP_2.class`), it is not matching `@ValidateConditional's` `` `targetGroup `` attribute(`GROUP_1.class`).

Specifying a Validation Group

As seen in the previous example, there may be many params decorated with *javax.validation.constraints*. The framework needs to uniquely identify which validations should be

applied when the `when` condition is **true**. To handle this, the `groups` attribute will be used as it is supplied by of all *javax.validation.constraints* annotation classes. The final subset of params where validations will be applied will be composed of only those whos `groups` attribute contains the `@ValidateConditional targetGroup` attribute.

`@ValidateConditional's `targetGroup` parameter is simply a marker interface of type `ValidationGroup` to be used for identification purposes by the framework to identify the subset of params.

NOTE	Use `@ValidateConditionals` when: • Different groups should be applied based on multiple *when* conditions • Multiple groups should be applied for the same *when* condition

Validation Group Identity Classes

In the previous example, we used `targetGroup = GROUP_1.class`. `GROUP_1.class` is an identity class that implements the `ValidationGroup` interface and is used by the framework to identify *javax.validation.constraints* annotatons that should be applied.

Any implementation that implements the *ValidationGroup* interface may be used as a group identity class. For convenience, a set of identify class implementations have been defined within `@ValidateConditional` as `GROUP_X.class`, where 0 <= **X** <= 29.

NOTE	If additional marker classes are needed, simply create a new implementation of `ValidationGroup` and use that class in the `targetGroup` attribute as well as the corresponding param's *javax.validation.constraints* annotation.

Controlling Scope

The framework has provides the ability to define a `ValidationScope` as part of `@ValidateConditional's `scope` to provide control over which params should be considered for conditional validation. The following scopes are available:

Table 3. @ValidateConditional Scopes

Name	Description
SIBLING	Applies validations to sibling params relative to the target param.
CHILDREN	Applies validations to children params relative to the target param.

NOTE	Scopes are recursive in that all nested params underneath params located in the scopes identified above would also have conditional validations applied.

Specifying Target Path

There may be a need to apply conditional validations to one or many different paths based on some `when` condition decorating a single param. The attribute `targetPath` can be used to target one or more paths to apply conditional validation to when a corresponding `when` condition is `true`. Consider the scenario below:

```
@Model @Getter @Setter
public static class Section1 {

    @NotNull(groups = { GROUP_1.class })
    @Calendar
    private String dateOfBirth;

    ...
}
```

```
@Model @Getter @Setter
public static class Section2 {

    @NotNull(groups = { GROUP_1.class })
    @Calendar
    private String neuterDate;

    ...
}
```

```
@ValidateConditional(when = "state == 'dog'", targetGroup = GROUP_1.class, targetPath
= { "../section1", "../section2" }, scope = ValidationScope.CHILDREN)
@TextBox(postEventOnChange = true)
private String petType;

private Section1 section1;

private Section2 section2;
```

In this scenario, the configuration for `petType` mandates that both `section1` and `section2` would be targetted and all children params of these sections (recursive) would have conditional validation applied if `GROUP_1.class` is present on the `Constraint` annotation.

In this case, both `dateOfBirth` and `neuterDate` would have conditional validation applied.

NOTE	Similar behavior could have been achieved by omitting the `targetPath` attribute and using `ValidationScope.SIBLING`. This was simply an example to portrait the flexibility of `targetPath` and `scope`.

ValuesConditional

`@ValuesConditional` is an extension capability provided by the framework. The annotation is used to conditionally set the `@Values` configuration for a dependent field based on a SpEL condition. This annotation can be triggered for multiple events. Framework provides default event handling for this annotation on StateChange and StateLoad.

Consider the following sample defined Values:

SampleValues.java

```
// Sample Values Implementations
public static class SRC_FOODS_ALL implements Source {
    @Override
    public List<ParamValue> getValues(String paramCode) {
        final List<ParamValue> values = new ArrayList<>();
        values.add(new ParamValue("Generic Food 1", "Generic Food 1"));
        values.add(new ParamValue("Generic Food 2", "Generic Food 2"));
        return values;
    }
}
public static class SRC_FOODS_DOG implements Source {
    @Override
    public List<ParamValue> getValues(String paramCode) {
        final List<ParamValue> values = new ArrayList<>();
        values.add(new ParamValue("Dog Food 1", "Dog Food 1"));
        return values;
    }
}
public static class SRC_FOODS_CAT implements Source {
    @Override
    public List<ParamValue> getValues(String paramCode) {
        final List<ParamValue> values = new ArrayList<>();
        values.add(new ParamValue("Cat Food 1", "Cat Food 1"));
        return values;
    }
}
```

Given a defined set of Values that we can assign using the @Values annotation, we can explicitly define conditions to set a dependent field's values.

ValuesConditional.java

```
@Model
@Getter @Setter
public static class StatusForm {

    @ValuesConditional(target = "../petFoodSelection", condition = {
            @Condition(when = "state=='dog'", then = @Values(SRC_FOODS_DOG.class)),
            @Condition(when = "state=='cat'", then = @Values(SRC_FOODS_CAT.class)),
        }
    )
    @TextBox(postEventOnChange = true)
    private String petType;

    @Radio
    @Values(SRC_FOODS_ALL.class)
    private String petFoodSelection;
}
```

In this example, *petType* is the field and *petFoodSelection* is the dependent field. We set *petFoodSelection* to contain the defaults ["Generic Food 1", "Generic Food 2"] initially and conditionally define those values when *petType*'s state is "dog" or "cat".

When the state of *petType* is "dog", then the Values for *petFoodSelection* will be ["Dog Food 1"]. Similarly when the state of *petType* is "cat", then the Values for *petFoodSelection* will be ["Cat Food 1"].

Conceptually speaking, we are **pushing** the updates of Values to the dependent field whenever the state of the annotated field loads or is changed.

VisibleConditional

`@VisibleConditional` is an extension capability provided by the framework. The annotation is used to conditionally set the UI visibility of a param based on a SpEL condition. This annotation can be triggered for multiple events. Framework provides default event handling for this annotation on StateChange and StateLoad.

VisibleConditional.java

```
@VisibleConditional(when="state == 'hooli'", targetPath="../p2")
private String p1;

private String p2;
```

4.1.2. Core Config Annotations

ConceptId

ConceptId.java

```
@TextBox(postEventOnChange = true)
@ConceptId(value = "IOT1.1.1")
@Label(value = "If Other, provide reason")
private String otherReason;
```

Config

Config.java

```
@Config(url="/pageAddEditGoal/tileEditGoal/sectionEditGoal/goalDetailsForm/_nav?pageId
=pageCarePlanSummary")
@Button(type = Button.Type.PLAIN)
private String cancel;
```

A class with @Config annotation is used to perform an action on button click. In most cases, the action is to retrieve values via HTTP Rest calls from database (MongoDB), and display on the web page.

In the example shown above, when the button is clicked, the control will be navigated to the specified url.
nav is the http call for navigation.

The possible Actions are: -

- get for HTTP GET
- new for HTTP post
- update for HTTP update
- delete for HTTP delete
- search for searching
- nav for navigation
- process for custom process/ work-flow definitions

ConfigNature

Ignore

Framework persists the data objects of a class in database using @Repo by serializing the class and associating a version number that is called seriaVersionUID. However, if we do not want the framwework to serialize for time being, we can use @Ignore component of ConfigNature class. The following example shows that: -

StartsWith.java

```
@Domain(value="patient", includeListeners={ListenerType.persistence})
@Repo(value=Database.rep_mongodb, cache=Cache.rep_device)
@ToString
public class Patient extends IdString {

    @Ignore
    private static final long serialVersionUID = 1L;

    }
```

Domain

Core Config configuration @Domain annotation persists data.

Core config @Domain will always be followed by @Repo that will specify the way data is persisted.

includeListeners={ListenerType.persistence, ListenerType.update} of @Domain specifies that the data will be persisted.
value=Database.rep_mongodb of @Repo specifies that a class with @Domain annotation will use MongoDb for persistence.

Domain.java

```
@Domain(value="cmcase", includeListeners={ListenerType.persistence,
ListenerType.update})
@Repo(value=Database.rep_mongodb, cache=Cache.rep_device)
public class CMCase extends IdString {
}
```

NOTE	Please read @Repo for mroe information regarding @Repo annotation.

DomainMeta

Similar to ViewStyle component, DomainMeta component is used to define an ANNOTATION_TYPE level annotation that is used with few view config annotations, as follows: -

- @ConceptId

```
@Retention(RetentionPolicy.RUNTIME)
    @Target(value={ElementType.ANNOTATION_TYPE})
    @Inherited
    public @interface DomainMeta {

    }
```

Here is an example: -

```
@Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.FIELD})
    @DomainMeta
    public @interface ConceptId {
        String value() default "";
    }
```

Execution

Execution is inherited annotation for @Config and @KeyValue. It is not currently being directly used, but is there for hieracrchial purposes.

More documentation will be added here if Execution expands or is directly used.

Initialize

Initialize.java

```
@Section
@Initialize
@Config(url="/vpAdvancedMemberSearch/vtAdvancedMemberSearch/vsMemberSearchCriteria/vfA
dvMemberSearch/_process?fn=_setByRule&rule=updateadvmbrsearchcriteria")
private VSMemberSearchCriteria vsMemberSearchCriteria;
```

MapsTo

Path

Path.java

```
@Path()
private Long version;
```

Type

Type.java

```
@MapsTo.Type(CMCase.class)
public static class SectionEditGoal  {
    }
```

NOTE	If it is not mapped *(@Mapped)*, an exception will be thrown. If no exception is thrown, defaults to silent.

Model

`@Model` is a marker annotation for the framework to acknoledge an class declaration as a *model*

entity. Useful when created "nested entities".

Model is similar to Domain, except that domain is seen as the "root" entity, or the topmost parent in a nested entity. Typically, model is expected to be used on nested class declarations. See the following example:

Parent.java

```
@Domain("parent")
@Getter @Setter @ToString
public class Parent {

    private Child1 child;
}
```

Child1.java

```
@Model
@Getter @Setter @ToString
public class Child1 {

    private Child2 child;
}
```

Child2.java

```
@Model
@Getter @Setter @ToString
public class Child2 {

}
```

In this example, the object hierarchy is `Parent` > `Child1` > `Child2`, where `Child1` and `Child2` are decorated with `@Modal`.

ParamContext

@ParamContext is used to set the contextual properties of a field during the *OnStateLoad* event (e.g. *visible*, *enabled*).

The intent of **@ParamContext** is to be able to decorate fields to define default contextual behavior. For example:

SampleView.java

```
public static class SampleView {

   @TextBox
   @ParamContext(enabled=false, visible=false)
   private String myTextBox;
}
```

In this scenario we have configured the contextual values for *enabled* and *visible* to be **false** for *myTextBox*. These values will be set during the *OnStateLoad* event and *myTextBox* consequently will not be enabled or visible when the corresponding page is rendered.

TIP	**@ParamContext** can also be defined on annotations. In these scenarios when a field is decorated with that annotation, then the handler for **@ParamContext** will execute. This may be useful when building a framework extension.

PrintConfig

Defines print configuration that should be applied when invoking the print action on a view component.

Decoratable Fields

`@PrintConfig` currently is currently supported for the following components:

Button

Sample Configuration

```
@Button(style = Button.Style.PRINT, printPath = "/domain/page/tile/modal")
@PrintConfig(autoPrint = false)
private String print;
```

The previous example opens a print dialog with the rendered HTML content resolved from the `@Button printPath`, or the content rendered from `"/domain/page/tile/modal"`, in a new window/tab. Because `autoPrint = false` is given, the new window/tab will not close after the print dialog is closed (default behavior). There are a number of different configurable options when using `@PrintConfig`, such as customized print styles. Please review the Javadocs for `@PrintConfig` for more details.

TIP	`@PrintConfig` values may need to be different under different scenarios. In that case, define multiple view config fields (e.g. another `@Button`) and conditionally hide or show the print button that performs the desired functionality.

SearchNature

StartsWith

This component is ued to validate a field. The wilcard attribute determines the validation criteria for a field. This is a server side component.

StartsWith.java

```
@NotNull
@StartsWith
@Label(value = "First Name")
private String firstName;
```

NOTE	The example will always search the first name that starts with anything, represented with the default value. A specific search criteria can be specified using wildcard attribute of @StartsWith.

Repo

@Repo is used to determine where the data will be persisted. It is always used along with @Domain.

rep_mongodb

The following example shows how data is persisted/ retrieved using MongoDB as a source.

Repo.java

```
@Domain(value="cmassessment", includeListeners={ListenerType.persistence})
@Repo(alias="cmassessment",value=Database.rep_mongodb, cache=Cache.rep_device)
@Getter @Setter
public class CMAssessment extends IdString {
}
```

rep_ws

Database values can be persisted/ retrieved not only using internal MongoDb as a source, we can now use an external web service for the same purpose. All we have to do is create a REST call to a web service that will provide or store the data. Following is an example: -

ExtClient.java

```
@Domain(value="ext_client")
@Repo(value=Database.rep_ws, cache=Cache.rep_device)
@Getter @Setter @ToString(callSuper=true)
public class ExtClient {

}
```

Notice **@Repo(value=Database.rep_ws)** in the code above. This indicates that it will make use of

an external web service call rather than a MongoDb call for ExtClient data objects' persistence/ retrieval purposes.

The advantage of this feature is that now we do not rely on internal source only. This gives us better flexibility and maintainability

Rule

@Rule allows its decorated field a mechanism for triggering one or more rule definitions during its OnStateLoad and OnStateChange events.

SampleRuleEntity.java

```
@Domain(value="sample_rule_entity", includeListeners={ListenerType.persistence})
@Repo(Database.rep_mongodb)
@Getter @Setter
public class SampleRuleEntity {

    // Execute the rule at "rules/sample_increment" during the OnStateLoad and
    // OnStateChange events of ruleParam.
    @Rule("rules/sample_increment")
    private String rule_param;
}
```

By default, the framework provides support for firing all rules for a given domain entity. That is, for the **SampleRuleEntity.java** above we might have a rule file defined as **sample_rule_entity.drl** which will be automatically fired by naming convention.

For cases where additional configuration for other rules is needed, @Rule can be used.

Values

Values provides a mechanism for populating a fields *values* property. This can be used by a number of components to perform such functions as: define a set of selections for radio buttons and checkboxes, or populating a dropdown list.

Source

The Source is a simple abstraction for providing a contract between implementations to provide data to the framework.

Source is exclusively used for **@Values**.

Source.java

```
public static interface Source {
    public List<ParamValue> getValues(String paramCode);
}
```

We can use this to define several different types of values providers. A simple static Source

implementation is shown below:

SampleStaticSource.java

```
public class SampleStaticSource implements Source {
    public List<ParamValue> getValues(String paramCode) {
        List<ParamValue> values = new ArrayList();
        values.add(new ParamValue("sample.value.1", "Sample Value 1"));
        return values;
    }
}
```

EnableAPIMetricCollection

@EnableAPIMetricCollection allows its decorated class, method level logging mechanism with log entries for method - **entry**, **exit**, **arguments** and **response**. It can be configured with log level of *info* or *debug* for arguments and response.

EnableAPIMetricCollection.java

```
@EnableAPIMetricCollection(args=LogLevel.info,resp=LogLevel.info)
@Getter(value=AccessLevel.PROTECTED)
public class SampleMetricLog {

    public Object calculate(String input) {
        // some logic
    }
}
```

By Default, the logging level for arguments & response is *debug* and *info* for entry and exit. Also, method exit logs the execution time of the method in milliseconds.

Client applications can annotate the custom config beans with @EnableAPIMetricCollection to enable a similar logging mechanism.

Table 4. EnableAPIMetricCollection Attributes

Name	Type	Default	Description
args	LogLevel	debug	LogLevel for logging method arguments
resp	LogLevel	debug	LogLevel for logging method response

4.1.3. View Config Annotations

Accordion

Accordion groups a collection of contents in tabs.

Allowed Parent Components

Form, Section

Allowed Children Components

AccordionTab

Sample Configuration

```
@Accordion
private VAQuestionnaire vaQuestionnaire;

@Model
@Getter @Setter
public static class VAQuestionnaire {

    @AccordionTab
    private VATTab1 vatTab1;

    @AccordionTab
    private VATTab2 vatTab2;

    @Model
    @Getter @Setter
    public static class VATTab1 { ... }

    @Model
    @Getter @Setter
    public static class VATTab2 { ... }
}
```

AccordionTab

AccordionTab contains a collection of contents for an individual section within an Accordion.

Allowed Parent Components

Accordion

Allowed Children Components

Accordion Tab supports all of the same children that are supported as children by Form.

Sample Configuration

```
@AccordionTab
private VATTab1 vatTab1;

@Model
@Getter @Setter
public static class VATTab1 {

    @TextBox
    private String txt1;

    @ButtonGroup
    private VBGMain vbgMain;
}
```

ActionTray

ActionTray is an extension to the standard button input element with icons and theming.

Allowed Parent Components

Domain

> **NOTE** ActionTray can only be contained within a View Layout definition.

Allowed Children Components

Button

Sample Configuration

```
@ActionTray
private VSActionTray vsActionTray;

@Model
@Getter @Setter
public static class VSActionTray {

    @Label(value = " ")
    @Config(url = "...")
    @Button(imgSrc = "fa-id-card", cssClass = "icon btn btn-icon mr-0", title =
"Record a Contact")
    private String contactRecord;
}
```

Button

Button is an extension to the standard button input element with icons and theming.

Allowed Parent Components

ActionTray, ButtonGroup, Form, Grid, Section

Allowed Children Components

None. `@Button` should decorate a field having a simple type.

Sample Configuration

```
@Config(url = "...")
@Button(type = Button.Type.DESTRUCTIVE)
private String delete;
```

ButtonGroup

ButtonGroup contains a collection of Button components.

Allowed Parent Components

Form, Section

Allowed Children Components

Button

Sample Configuration

```
@ButtonGroup(cssClass="text-sm-right")
private VBGGoals vbgGoals;

@Model
@Getter @Setter
public static class VBGGoals {

    @Button
    private String btn1;

    @Button
    private String btn2;
}
```

Calendar

Calendar is an input component to select a date.

Allowed Parent Components

- Form

Allowed Children Components

None. `@Calendar` should decorate a field having a simple type of a supported date type:

- `LocalDate`
- `LocalDateTime`

- `ZonedDateTime`
- `Date`

Sample Configuration

```
@Path
@Calendar(postEventOnChange=true, showtime=true, hourFormat="24")
private LocalDate startDate;

@Path
@Calendar(postEventOnChange=true, timeOnly=true)
private LocalDateTime startDate;
```

CardDetail

CardDetail is a flexible container component.

Allowed Parent Components

Accordion, AccordionTab, CardDetailsGrid, Section

Allowed Children Components

CardDetail.Header, CardDetail.Body

Sample Configuration

```
@CardDetail(title="Member Overview", cssClass="contentBox right-gutter bg-alternate
mt-0")
private VCDMember vcdMember;

@Model
@Getter @Setter
public static class VCDMember {

    @CardDetail.Header
    VCDHMain header;

    @CardDetail.Body
    VCDBMain body;

    @Model
    @Getter @Setter
    public static class VCDHMain { ... }

    @Model
    @Getter @Setter
    public static class VCDBMain { ... }
}
```

NOTE	**contentBox right-gutter bg-alternate mt-0** overrides the default cssClass specified for the `@CardDetail` component.

CardDetail.Body

CardDetail.Body is a container component within CardDetail that contains the main content.

Allowed Parent Components

CardDetail

Allowed Children Components

FieldValue, Link, Paragraph, StaticText

Sample Configuration

```
@CardDetail.Body
VCDBMain body;

@Model
@Getter @Setter
public static class VCDBMain {

    @FieldValue
    private String value1;
}
```

CardDetail.Header

CardDetail.Header is a container component within CardDetail that contains the header content.

Allowed Parent Components

CardDetail

Allowed Children Components

ButtonGroup, FieldValue, Paragraph

Sample Configuration

```
@CardDetail.Header
VCDHMain header;

@Model
@Getter @Setter
public static class VCDHMain {

    @FieldValue
    private String value1;
}
```

CardDetailsGrid

CardDetailsGrid contains a collection of CardDetail components.

Allowed Parent Components

Accordion, Section

Allowed Children Components

A field decorated with `@CardDetailsGrid` should be a collection/array with a defined type. The defined type is treated as the definition for the row data in the rendered content. This is referred to as the *collection element type*. The allowed children components of the collection element's type are:

CardDetail

Sample Configuration

```
@CardDetailsGrid(onLoad = true)
@Config(url = "...")
@Path(linked = false)
private List<ConcernsLineItem> vcdgConcerns;

@Model
@Getter @Setter
public static class ConcernsLineItem {

    @CardDetail
    private VCDMain vcdMain;

    @Model
    @Getter @Setter
    public static class VCDMain { ... }
}
```

CheckBox

Checkbox is an extension to standard checkbox element.

Allowed Parent Components

Form

Allowed Children Components

None. `@CheckBox` should decorate a field having a simple type of `Boolean` or `boolean`.

Sample Configuration

```
@CheckBox(postEventOnChange=true)
private boolean admin;
```

CheckBoxGroup

CheckBoxGroup is used for multi-select CheckBox components.

Allowed Parent Components

Form

Allowed Children Components

None. `@CheckBoxGroup` should decorate a field having a collection/array of type `String`.

Sample Configuration

```
@CheckBoxGroup(postEventOnChange=true)
private String[] days;
```

ComboBox

Combobox is used to select an item from a collection of options.

Allowed Parent Components

Form, Section

Allowed Children Components

None. `@ComboBox` should decorate a field having a simple type.

Sample Configuration

```
@Path
@Values(url = "...")
@ComboBox
private String goalCategory;
```

Supplying Values to a ComboBox

When decorated with `@Values`, the resolved values as defined by the logic within Values will be used as the options for the rendered `@ComboBox` component.

FieldValue

FieldValue is a container for displaying a single value. Has the ability to become an in place edit field with several configurable properties.

Allowed Parent Components

CardDetail.Body, CardDetail.Header, FieldValueGroup

Allowed Children Components

None. `@FieldValue` should decorate a field having a simple type.

Sample Configuration

```
@MapsTo.Type(Goal.class)
@Getter @Setter
public static class VCDHGoals {

    @FieldValue(showName = false, cols = "2")
    @Path
    private String description;

    @FieldValue(showName = false, iconField = "date")
    @Path
    private LocalDate targetDate;

    @FieldValue(showName = false, iconField = "planned")
    @Path("/status")
    private String status;

    @FieldValue(showName = false, datePattern = "MM/dd/yyyy")
    @Path
    private LocalDate startDate;
}
```

FieldValueGroup

FieldValue is a container for displaying a logical grouping of multiple FieldValue components.

Allowed Parent Components

CardDetail.Body

Allowed Children Components

FieldValue

Sample Configuration

```
@FieldValueGroup
private AddressGroup addressGroup;

@MapsTo.Type(Address.java)
@Getter @Setter
public static class AddressGroup {

    @FieldValue
    @Path
    private String line1;

    @FieldValue
    @Path
    private String line2;

    @FieldValue
    @Path
    private String city;

    @FieldValue
    @Path
    private String state;

    @FieldValue
    @Path
    private String zip;
}
```

FileUpload

FileUpload is an advanced uploader with dragdrop support, multi file uploads, auto uploading, progress tracking and validations.

Allowed Parent Components

Form

Allowed Children Components

None. `@FileUpload` should decorate a field having a simple type.

Sample Configuration

```
@FileUpload(type = "${app.config.upload.allowedTypes}")
private String uploader;
```

Form

A Form is an HTML form container for user input contents.

Allowed Parent Components

Section

Allowed Children Components

Accordion, Button, ButtonGroup, Calendar, CheckBox, CheckBoxGroup, ComboBox, FileUpload, FormElementGroup, Grid, Header, Paragraph, PickList, Radio, Signature, TextArea, TextBox

Sample Configuration

```
@Form(cssClass = "twoColumn")
@Path(linked = false, state = State.External)
private VFGoals vfGoals;

@Model
@Getter @Setter
public static class VFGoals { ... }
```

FormElementGroup

TODO

Allowed Parent Components

TODO

Allowed Children Components

TODO

Sample Configuration

```
TODO
```

Grid

Grid is a table container capable of displaying tabular data.

Allowed Parent Components

Form, Section

Allowed Children Components

`@Grid` should decorate a field having a collection/array with a defined type. The defined type is treated as the definition for the row data in the rendered HTML table. This is referred to as the *collection element type*. The allowed children components of the collection element's type are:

GridColumn, LinkMenu, GridRowBody

Sample Configuration

```
@Grid
private List<PetLineItem> vgPets;

@Model
@Getter @Setter
public static class PetLineItem {

    @GridColumn
    private String field1;

    @LinkMenu
    private VLMMain vlmMain;

    @GridRowBody
    private ExpandedRowContent expandedRowContent;

    @Model
    @Getter @Setter
    public static class VLMMain { ... }

    @Model
    @Getter @Setter
    public static class ExpandedRowContent { ... }
}
```

Handling Multiple Row Selection in @Grid

Grid offers the ability to select multiple rows of data and submit the indexes of the selected rows to the server for processing.

In the example below, we can see an example of configuring a grid that should allow the user to select multiple rows and then perform some action with those selections.

```
@Grid(onLoad = true,
        rowSelection = true,
        postButtonUrl = "/view/page/tile/section/actionRemove",
        postButton = true,
        postButtonTargetPath = "temp_ids",
        postButtonAlias = "Remove")
@Path("/members")
@Config(url = "/p/member/_search?fn=example")
private List<VGGroupView> vgGroupList;

@Config(url = "/page/tile/section/tempIdsWrapper/_replace")
@Config(url = "/p/member:<!col!>/_delete", col = "<!/tempIdsWrapper/temp_ids!>"
private String actionRemove;

private TempIdsWrapper tempIdsWrapper;

@Model
@Getter @Setter
public static class TempIdsWrapper {
    private List<String> temp_ids;
}
```

GridColumn

GridColumn is a container for displaying a single value within a Grid.

Allowed Parent Components

Grid

Allowed Children Components

None. `@GridColumn` should decorate a field having a simple type.

Sample Configuration

```
@Model
@Getter @Setter
public static class PetLineItem {

    @GridColumn
    private String field1;
}
```

GridRowBody

GridRowBody is used to display additional content about the row data within a Grid.

Allowed Parent Components

Grid

Allowed Children Components

`@GridRowBody` will display children components in the same manner as Section does. See the *Allowed Children Components* of Section for more details.

Sample Configuration

```
@MapsTo.Type(Pet.class)
@Getter @Setter
public static class PetLineItem {

    @GridColumn
    @Path
    private String name;

    @GridRowBody
    private ExpandedRowContent expandedRowContent;

    @Model
    @Getter @Setter
    public static class ExpandedRowContent {

        @CardDetail
        private CardDetails cardDetails;
    }

    @Model
    @Getter @Setter
    public static class CardDetails {

        @CardDetail.Body
        private CardBody cardBody;
    }

    @Model
    @Getter @Setter
    public static class CardBody {

        @FieldValue
        @Path
        private String id;
    }
}
```

Header

Header is a container with a text header, equivalent to an HTML header.

Allowed Parent Components

Form

Allowed Children Components

None. `@Header` should decorate a field having a simple type.

Sample Configuration

```
@Header(size = Header.Size.H1)
private String header;
```

Hints

Hints is a UI styling behavior applied for some components. The behavior applied is delegated to each individual component where `@Hints` is supported.

Hints.java

```
@Link(url = "/#/h/cmdashboard/vpDashboard", imgSrc = "anthem-rev.svg")
@Hints(AlignOptions.Left)
@PageHeader(Property.LOGO)
private String linkHomeLogo;
```

Image

An image component used for displaying images.

Allowed Parent Components

None.

Allowed Children Components

None. `@Image` should not decorate a field and will typically be used as an annotation attribute in other components.

Sample Configuration

```
@MenuPanel(imgType = Image.Type.SVG, imgSrc = "notesIcon")
private VMPVisits vmpVisits;
```

Label

Label is used to display label text for a view component.

@Label can be used on all View annotations (e.g. Page, Tile, TextBox, etc.), wherever content is necessary. Multiple `@Label` annotations can be provided for managing content for different locales.

Sample Configuration

```
@Label("First Name")
private String firstName;

@Label(value="Last Name")
private String lastName;

@Label(value = "Select 3" , helpText = "Please select atleast 3 cities")
@CheckBoxGroup
private String[] city;

@Tile
@Label("Dashboard")
private VTDashboard vtDashboard;

@Page
@Label(" ")
private VPDashboard vpDashboardWithEmptyLabel;

@Label(value="Test Label C in English", helpText="some tooltip text here C")
@Label(value="Test Label A in French", localeLanguageTag="fr")
private String staticText;

@Textbox
private String addressline;
```

Link

Link is a hyperlink component used for navigation or user interaction of displayed text.

Allowed Parent Components

CardDetail, Grid, LinkMenu, Menu, Section

Allowed Children Components

None. `@Link` should decorate a field having a simple type.

Sample Configuration

```
// UI Navigation to page vpHome under domain petclinic
@Link(url = "/h/petclinic/vpHome")
private String linkToHome;

// Executes a request against _subscribe, and sets the image
@Link(url = "/notifications/_subscribe:{id}/_process", b="$executeAnd$configAnd$nav",
method = "POST")
private String subscribeToEmailNotifications;

// Creates an external link, as in a link navigating outside of the context of the
Nimbus framework.
@Link(url = "https://www.mywebsite.com", value = Link.Type.EXTERNAL, target = "_new",
rel = "nofollow")
private String myWebsiteLink;
```

LinkMenu

LinkMenu is a dropdown component used for displaying links within a Grid row.

Allowed Parent Components

Grid

Allowed Children Components

Link

Sample Configuration

```
@LinkMenu
private VLMMain vlmMain;

@Model
@Getter @Setter
private static class VLMMain {

    // Display a link that performs some @Config action
    @Label(value = "View Case")
    @Link
    @Config(url = "...")
    private String viewCase;

    // Display a link that performs some @Config action and uses an image icon.
    @Label(value = "Assign Case Owner")
    @Link(imgSrc = "task.svg")
    @Config(url = "...")
    private String assignCase;

    // Display a link that performs an external navigation in a new tab and passing a
dynamic value
    @Link(value = Type.EXTERNAL, target = "_blank", url =
"${app.endpoints.documents}/showDocument.aspx?documentid={documentId}")
    @Path(linked = false)
    private String viewDoc;

    private String documentId;
}
```

Menu

Menu is a container intended to display other navigation components.

Allowed Parent Components

Section

Allowed Children Components

Link

Sample Configuration

```
@Menu
private SideMenu sideMenu;

@Model
@Getter @Setter
public static class SideMenu {

    @Link(url = "#")
    private String home;

    @Link(url = "#")
    private String signup;
}
```

MenuLink

MenuLink is nestable menu link component that is exclusively used with MenuPanel.

Allowed Parent Components

MenuPanel

Allowed Children Components

None. `@MenuLink` should decorate a field having a simple type.

Sample Configuration

```
@Label("Link 1")
@MenuLink
private String link1;

@Label("Link 2")
@MenuLink
private String link2;
```

MenuPanel

MenuPanel is a hybrid of accordion-tree components.

Allowed Parent Components

MenuPanel, Page

Allowed Children Components

MenuLink, MenuPanel

Sample Configuration

```
@MenuPanel
private VMPView vmpView;

@Model
@Getter @Setter
public static class VMPView {

    @Label("Show View...")
    @MenuLink
    private String showView;

    @Label("Appearance")
    @MenuPanel
    private VMPAppearance appearance;

    @Model
    @Getter @Setter
    public static class VMPAppearance {

        @Label("Toggle Full Screen")
        @MenuLink
        private String toggleFullScreen;

        @Label("Toggle Always on Top")
        @MenuLink
        private String toggleAlwaysOnTop;
    }
}
```

Modal

Modal is a container to display content in an overlay window.

Allowed Parent Components

Tile

Allowed Children Components

Section

Sample Configuration

```
@Label("Modal Title")
@Modal
private VMMyModal vmMyModal;

@Model
@Getter @Setter
public static class VMMyModal {

    @Section
    private VSMain vs;

    @Model
    @Getter @Setter
    public static class VSMain { ... }
}
```

Displaying @Modal on Load

`@Modal` makes use of `@ParamContext` to render it's visible property to `false` during initialization by default. This and other similiar behaviors can be overridden by supplying the `context` attribute of `@Modal` with a `@ParamContext`.

```
@Modal(context = @ParamContext(enabled = true, visible = true))
private VMMyModal vmMyModal;

public static class VMMyModal { ... }
```

See ParamContext for more details.

Page

Page is a container component that groups a collection of contents.

Allowed Parent Components

Domain

Allowed Children Components

Tile

Sample Configuration

```
@Page(defaultPage = true)
private VPHome vpHome;

@Page
private VPPets vpPets;

@Model
@Getter @Setter
public static class VPHome {

    @Tile
    private VTMain vtMain;

    @Model
    @Getter @Setter
    public static class VTMain { ... }
}

@Model
@Getter @Setter
public static class VPPets { ... }
```

PageHeader

TODO

Allowed Parent Components

TODO

Allowed Children Components

TODO

Sample Configuration

```
@PageHeader(PageHeader.Property.USERNAME)
private String fullName;
```

Paragraph

Paragraph is a container for displaying text content.

Allowed Parent Components

Form, Header, Section

Allowed Children Components

None. `@Paragraph` should decorate a field having a simple type.

Sample Configuration

```
@Paragraph("Hello! Welcome to Pet Clinic.")
private String welcome;
```

PickList

PickList is used to reorder items between different lists.

Allowed Parent Components

Form

Allowed Children Components

PickListSelected

Sample Configuration

```
@PickList(sourceHeader = "Available Category", targetHeader = "Selected Category")
@Values(value = SomeCategory.class)
private PicklistType category;

@MapsTo.Type(SomeClass.class)
@Getter @Setter
public static class PicklistType {

    @PickListSelected(postEventOnChange = true)
    @Path("category")
    @Values(value = SomeCategory.class)
    @NotNull
    private String[] selected;
}
```

Using @PickList with Dynamic Values

The `@PickList` implementation is works closely with Values. In the previous sample configuration `@PickList` and `@PickListSelected` are using the same `@Values` source. If the values assigned to the `@PickList` component need to be conditionally set, meaning that the values on the source list are able to change, ValuesConditional can be used. See the following example:

```
@ComboBox(postEventOnChange = true)
@Values(value = petType.class)
@ValuesConditional(target = "../category", condition = {
    @ValuesConditional.Condition(when = "state == 'Dog'", then = @Values(value =
DogCategory.class)),
    @ValuesConditional.Condition(when = "state == 'Cat'", then = @Values(value =
CatCategory.class))
})
@VisibleConditional(targetPath = { "../category" }, when = "state != 'Horse'")
@EnableConditional(targetPath = { "../category" }, when = "state != 'Parrot'")
private String type;

@PickList(sourceHeader = "Available Category", targetHeader = "Selected Category")
private PicklistType category;

@MapsTo.Type(SomeClass.class)
@Getter @Setter
public static class PicklistType {

    @PickListSelected(postEventOnChange = true)
    @Path("category")
    @Values(value = AllCategory.class)
    @NotNull
    private String[] selected;
}
```

One caveat to this implementation is that the `@PickListSelected` component should have it's values set to the **comprehensive set** of possible values that the parent source list could contain. (e.g. The values in `DogCategory.class` and `CatCategory.class` should be contained in `AllCategory.class`). Not doing so may result in undesirable results.

PickListSelected

PickListSelected is a required supplementary configuration for PickList. The `@PickList` works exclusively with `@PickListSelected` by providing the UI a server-side location to store the *selected* values and any other metadata associated with those *selected* values.

Allowed Parent Components

PickList

Allowed Children Components

None. `@PickListSelected` should decorate an array or collection.

Sample Configuration

```
@PickListSelected(postEventOnChange = true)
@Path("category")
@Values(value = AllCategory.class)
@NotNull
private String[] selected;
```

Radio

Radio is an extension to standard radio button element.

Allowed Parent Components

Form

Allowed Children Components

None. `@Radio` should decorate a field having a `Boolean` or `boolean` value.

Sample Configuration

```
@Label("Some question:")
@Radio(postEventOnChange = true, controlId = "27")
@Values(url="~/client/orgname/staticCodeValue/_search?fn=lookup&where=staticCodeValue.
paramCode.eq('/q14')")
@Path
private String q14;
```

Section

Section is a container component that groups a collection of contents.

Allowed Parent Components

Grid, Modal, Section, Tile

Allowed Children Components

Accordion, Button, ButtonGroup, CardDetail, CardDetailsGrid, ComboBox, Form, Grid, Link, Menu, Paragraph, StaticText, TextBox

Sample Configuration

```
@Section
private VSMain vsMain;

@Model
@Getter @Setter
public static class VSMain { ... }
```

Signature

Signature is an HTML canvas element that can be used to capture signature content. The `@Signature` component will render the following on the UI:

- *canvas* on which the user can draw
- *Clear* button, which will clear the *canvas*
- *Accept* button, which will store the user-drawn content captured in *canvas* as a *data-url* representation
- <*img*>, which contains the stored image after clicking *accept* button

Allowed Parent Components

Form

Allowed Children Components

None. `@Signature` should decorate a field having a simple type.

Sample Configuration

```
@Signature(postEventOnChange = true)
private String employeeSignature;
```

StaticText

StaticText is a container for displaying html content or text "as is" to the UI. `@StaticText` may be used to bind unsafe HTML directly onto the page.

> With great power comes great responsibility.
>
> — Voltaire

Allowed Parent Components

CardDetail, Section

Allowed Children Components

None. `@StaticText` should decorate a field having a simple type.

Sample Configuration

```
@StaticText
private String description;
```

TextArea

TextArea is a text input component that allows for a specified number of rows.

Allowed Parent Components

Form

Allowed Children Components

None. `@TextArea` should decorate a field having a simple type.

Sample Configuration

```
@TextArea
private String description;
```

TextBox

TextBox is a text input component.

Allowed Parent Components

Form, Section

Allowed Children Components

None. `@TextBox` should decorate a field having a simple type.

Sample Configuration

```
@TextBox
private String sampleTextField;
```

Tile

Tile is a container component that groups a collection of contents.

Allowed Parent Components

Page

Allowed Children Components

Header, Modal, Section, Tile

Sample Configuration

```
@Tile(size = Tile.Size.Large)
private VTMain vtMain;

@Model
@Getter @Setter
public static class VTMain { ... }
```

TreeGrid

TreeGrid is used to display hierarchical data in tabular format.

Allowed Parent Components

Section, Form

Allowed Children Components

`@TreeGrid` should decorate a field having a collection/array with a defined type. The defined type is treated as the definition for the row data in the rendered HTML table. This is referred to as the *collection element type*. The allowed children components of the collection element's type are:

GridColumn, LinkMenu, TreeGrid

Sample Configuration

```
@TreeGrid
@Path(linked = false)
@Config(url = "<!#this!>/.m/_process?fn=_set&url=/p/pethistory/_search?fn=example")
private List<PetHistoryLineItem> treegrid;

@Model
@Getter @Setter
public static class PetHistoryLineItem { ... }
```

TreeGrid

TreeGridChild is the recursive child of TreeGrid and is used to display hierarchical data in tabular format.

Allowed Parent Components

TreeGrid

Allowed Children Components

`@TreeGridChild` should decorate a field having a collection/array with a defined type. The defined type is treated as the definition for the row data in the rendered HTML table. This is referred to as the *collection element type*. The type should **ALWAYS** match the collection element type of the parent field decorated with `@TreeGrid` and consequently, the allowed children are the same as the allowed children of TreeGrid.

Sample Configuration

```
@TreeGrid
@Path(linked = false)
@Config(url = "<!#this!>/.m/_process?fn=_set&url=/p/pethistory/_search?fn=example")
private List<PetHistoryLineItem> treegrid;

@MapsTo.Type(PetHistory.class)
@Getter @Setter
public static class PetHistoryLineItem {

    @GridColumn
    @Path
    private String name;

    @TreeGridChild
    @Path
    private List<PetHistoryLineItem> children;
}
```

ViewRoot

ViewRoot is the entry point for a view domain definition.

Allowed Parent Components

None.

Allowed Children Components

Page

ViewRoot.java

```
@Domain(value = "sampleview", includeListeners = { ListenerType.websocket }, lifecycle
= "sampleview")
@MapsTo.Type(SampleView.class)
@Repo(value = Database.rep_none, cache = Cache.rep_device)
@ViewRoot(layout = "sampleviewlayout")
@Getter @Setter
public class VRSampleView { ... }
```

4.2. Conditional Configuration Examples

The following samples showcase how to apply common conditional scenarios using the framework's *@Conditional* annotations.

Make Textbox "Read-Only" by Default

```
@TextBox
@ParamContext(enabled = false, visible = true)
private String textbox;
```

Make Textbox "Read-Only" When a Question is Answered as "Yes"

```
@TextBox(postEventOnChange = true)
@EnableConditional(when = "state == 'Yes'", targetPath = { "/../textbox" })
private String condition;

@TextBox
private String textbox;
```

Make Textbox Mandatory by Default

```
@TextBox
@NotNull
private String textbox;
```

Make Textbox Mandatory When a Question is Answered as "Yes"

```
@TextBox(postEventOnChange = true)
@ValidateConditional(when = "state == 'Yes'", targetGroup =
ValidationGroup.GROUP_1.class })
private String condition;

@TextBox
@NotNull(groups = { ValidationGroup.GROUP_1.class })
private String textbox;
```

Make Textbox Validate by Pattern and be Mandatory When a Question is Answered as "Yes"

```
@TextBox(postEventOnChange = true)
@ValidateConditional(when = "state == 'Yes'", targetGroup =
ValidationGroup.GROUP_1.class })
private String condition;

@TextBox
@NotNull(groups = { ValidationGroup.GROUP_1.class })
@Pattern(regexp = "Hello (.*)!", groups = { ValidationGroup.GROUP_1.class })
private String textbox;
```

Make Textbox Editable by Default

```
@TextBox
private String textbox;
```

Make Combobox Values Change and be Mandatory Based on Another Field

```
@TextBox(postEventOnChange = true)
@ValidateConditional(when = "state == 'Yes'", targetGroup =
ValidationGroup.GROUP_1.class })
private String condition;

@ComboBox(postEventOnChange = true)
@NotNull(groups = { ValidationGroup.GROUP_1.class })
@ValuesConditional(target = "../condition", condition = {
    @Condition(when = "state=='Yes'", then = @Values(YES_VALUES.class)),
    @Condition(when = "state=='No'", then = @Values(NO_VALUES.class))
})
private String combobox;
```

Make TextBox Editable or Not-Editable by Default When a Form Loads

```
// Editable by default
@TextBox
private String textbox1;

// Not Editable by default - Method 1 (Preferred)
@TextBox
@ParamContext(enabled = false, visible = true)
private String textbox2;

// Not Editable by default - Method 2
@TextBox
@EnableConditional(when = "false", targetPath = "../")
private String textbox2;
```

Make Textbox Editable and Apply Validations Based on the Value of Another Field

```
@TextBox(postEventOnChange = true)
@EnableConditional(when = "state == 'Yes'", targetPath = { "/../textbox" })
@ValidateConditional(when = "state == 'Yes'", targetGroup =
ValidationGroup.GROUP_1.class })
private String condition;

@TextBox
@NotNull(groups = { ValidationGroup.GROUP_1.class })
private String textbox;
```

Make Textbox Already Having Validations have Additional Validations Based on the Value of Another Field

```
@TextBox(postEventOnChange = true)
@EnableConditional(when = "state == 'Yes'", targetPath = { "/../textbox" })
@ValidateConditional(when = "state == 'Yes'", targetGroup =
ValidationGroup.GROUP_1.class })
private String condition;

@TextBox
@NotNull(groups = { ValidationGroup.GROUP_1.class })
@Min(10)
private String textbox;
```

Make Form/Section Disabled Based on Some Condition

```
@TextBox(postEventOnChange = true)
@EnableConditional(when = "state == 'Yes'", targetPath = { "/../form" })
private String condition;

@Form
private MyForm form;

@Model @Getter @Setter
public static class MyForm {
    // Form implementation here...
}
```

Make Textbox Hidden on Some Condition and On Another Condition, Make it Visible and Mandatory.

```
@TextBox(postEventOnChange=true)
@ActivateConditional(when = "state == 'No'", targetPath = { "../textbox" })
@ValidateConditional(when = "state == 'Yes'", targetGroup = GROUP_1.class)
private String condition;

@TextBox
@NotNull(groups = { GROUP_1.class })
private LocalDateTime textbox;
```

Make Textbox Mandatory When State is Anything but a Particular Value

```
@TextBox(postEventOnChange=true)
@ValidateConditional(when = "state != null && state != 'Yes'", targetGroup =
GROUP_1.class)
private String condition;

@TextBox
@NotNull(groups = { GROUP_1.class })
private String textbox;
```

Make Textbox "Read-Only" When State is Anything but a Particular Value

```
@TextBox(postEventOnChange=true)
@EnableConditional(when = "state != null && state != 'Yes'", targetGroup =
GROUP_1.class)
private String condition;

@TextBox
@NotNull(groups = { GROUP_1.class })
private String textbox;
```

Make Textbox Hidden When State is All but a Particular Value

```
@TextBox(postEventOnChange=true)
@ActivateConditional(when = "state!= null && state != 'Yes'", targetGroup =
GROUP_1.class)
private String condition;

@TextBox
@NotNull(groups = { GROUP_1.class })
private String textbox;
```

4.3. Values Configuration Examples

The following samples showcase how to apply common scenarios using the framework's *@Values* annotation.

Using a static Source implementation to define a set of values

```
@Values(SampleStaticSource.class)
@CheckBoxGroup
private String petTypes;
```

In this example, all of the values retrieved from SampleStaticSource.getValues will be displayed as a collection of checkboxes.

Using a url-based Source implementation to define a set of values

```
@Values(url="CLIENT_ID/ORG/p/staticCodeValue/_search?fn=lookup&where=staticCodeValue.p
aramCode.eq('/petType')")
@CheckBoxGroup
private String petTypes;
```

In this example, all of the values retrieved from the url defined in **@Values** will be displayed as a collection of checkboxes.

4.4. Release Notes

Framework release notes can be found here. The most recent release notes are located at the top.

1.1.10

This is 1.1.10 version of the framweork. This version includes the following changes:

- There is an addition of new annotation `@PrintConfig`
- An attribute named `cssClass` is added to `@GridColumn`
- Added support for printing container elements

For more detailed release notes, see the 1.1.10.x Release Notes

1.1.9

This is a minor bugfix/feature version of the Framework.

There is an addition of new feature `@GridColumn` to support cssClass for override in View Config Annotation

For more detailed release notes, see the

1.1.9.x Release Notes

1.1.8

This is a minor bugfix/feature version of the Framework, including only a few defect fixes. This version was cut and prematurely promoted from a staging artifact to a final artifact due to misunderstanding of the artifact promotion process in the Nexus Repository.

For more detailed release notes, see the 1.1.8.x Release Notes

1.1.7

Prior to this release, the release artifacts were hosted in the Anthem Inc. private Artifactory and hence were not made available to the general public. This release markes the first release that the framework has been released from the Github source code at https://github.com/openanthem/nimbus-core.

The majority of work in this and the previous unofficial releases can be seen in UI support. Just a few to mention are the addition of new components such as `@InputSwitch`, `@FormElementGroup` and `@PickList`. Another big victory has been in adding support for dynamic labels and styles to params, so that users may change the labels of virtually any component under a given situation.

Numerous refactors have taken place to improve code quality and efficiency, not to mention an A+ report card from sonarcloud.io.

Though not listed in these release notes, all of the changes mentioned in this release have been

added since the version 1.0.7.RELEASE.

For more detailed release notes, see the 1.1.x Release Notes

1.0.4.RELEASE

This is a minor bugfix/feature version of the Framework.

For more detailed release notes, see the 1.0.4.x Release Notes

1.0.3.RELEASE

This is a minor bugfix/feature version of the Framework.

For more detailed release notes, see the 1.0.3.x Release Notes

1.0.2.RELEASE

This release included a necessary fix for the Signature Component including a zoom-in/zoom-out feature. No other changes were made.

1.0.1.RELEASE

This release included a necessary fix for production logging support. No other changes were made.

1.0.0.RELEASE

This is the initial release of the Framework.

This release builds on top of many of the original concepts and designs introduced in the inception of the Framework. General support for core and view configuration is available for the most basic of components as well as production level logging support for identifying issues on the fly.

For more detailed release notes, see the 1.0.0.x Release Notes