Sample training:

**Annotation Config**

The Framework offers several different means of configuration in the application through Java based annotations.

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, @Button, @Grid, @Page, @Tile, @Section, @Calendar, @Accordion)

• **Core Config Annotations** - Execute functional instructions or commands within the Framework.

Ex: @Config(url="~/client/org\_name/\_nav?pageId=vpAdvancedCaseSearch")

**• Conditional Config Annotations** - Execute common functional instructions based on Boolean SpEL expressions.

Ex:

@EnableConditional(when="state == 'hooli'", targetPath="../enable\_p2")

private String enable\_p1;

private String enable\_p2;

**Domain Entity**:

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.

**Config Annotations :** ex: (@Domain, @Model,@Repo,..etc)

Config Annotations will be used to instruct the Framework on how to handle the domain entities created from this domain entity during certain events.

@Domain("person")

@Getter @Setter @ToString

public class Person {

private String firstName;

private String lastName;

}

Person has been defined as a simple domain entity with getters and setters defined for each of it’s fields

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

1)@Domain,

2)@Model

**Defining Domain Behavior**:

@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.

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.

@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;

}

The Framework is responsible 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.

To take full advantage of abstraction capabilities within the Framework, the domain entities are essentially created in the same fashion. i.e We have **two types of Domain Entities**:

core domain entity and a view domain entity.

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

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

**Defining a View**

The UI Framework expects a specific pattern in terms of where components are placed.

for setting up the standard page view.ex: (Root/Page/Tile/Section/Form..)

View Config Annotations has explicitely defined which components are to be placed where.

Ex: The following is suggested:

• @Domain/@ViewRoot -

• @Page -

• @Tile -

• @Section -

• @Form -

--

**Ex: code:**

@Domain(value = "myView", includeListeners = { ListenerType.websocket })

@Repo(value = Repo.Database.rep\_none, cache = Repo.Cache.rep\_device)

@ViewRoot(layout = "")

@Getter @Setter

public class EmployeeRoot {

@Page(default = true)

private EmployeePage1 employeePage1;

@Page

private EmployeePage2 employeePage2;

@Model

@Getter @Setter

public static class EmployeePage1 {

@Tile

private EmployeeTile1 employeeTile1;

}

@Model

@Getter @Setter

public static class EmployeeTile1 {

@Section

private EmployeeSection1 employeeSection1;

}

@Model

@Getter @Setter

public static class EmployeeSection1 {

@Form

private EmployeeForm1 employeeForm1;

}

@Model

@Getter @Setter

public static class EmployeeForm1{

@TextBox

private String firstName;

}

@Model

@Getter @Setter

public static class EmployeePage2 {

@Tile

private VPTile tile2;

}

……

}

---

**@Model** is a marker annotation for the framework to acknowledge a class declaration as a model entity. Useful when created "nested entities".(as shown in above class)

i.e Model is similar to Domain, except that domain is seen as the "root" entity, or the topmost parent in a nested entity.

Frame work provided a DomainConfigBuilder to scan the register Domain entities, and load its information while applicaton startup.

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.yml file.

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

ex:

application.yml

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.

--

After application startup The instace of the registered Domain entity is treated as a Param by framework.

The Param is an interface provided by framework , The Param interface defines a number of properties on it(respective domain Entity) 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.

**UI/View:**

UI of an application can be created using the framework by by simply using the View Config Annotations and Conditional Config Annotations;

but, when needing to manipulate the model directly (perhaps in a rule or SpEL expression/using any custom handlers), having an understanding of Param will be extremely useful.

--

**Mapping Between core and View:**

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.

@mapsTO concept:

@mapsTO.Type on classlevel

@mapsTO.Path on paramI(property) level

Ex:

**Configuring a Mapping**

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 definition 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(“/name”)

@Label("Enter name: ")

private String personName;

}

}

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

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.

**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.

The path is represented as a URI address that uniquely identifies a param’s location within the Framework.

**Ex with Command Query DSL. :**

https://localhost:8080/cmproduct/client/org/app/p/viewCMLandingRoot/cmLandingPage/cmLandingTile/caseLandingSection/caseLandingForm/submitButton.

--

‘Protocol+HOST+PORT+client-code + app-code + /p + domain-alias + action + behavior+..

**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.

The Command Query DSL is represented in URL format (as shown above Command 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

Ex:

https://localhost:8080/cmproduct/client/org/app/p/viewCMLandingRoot/cmLandingPage/cmLandingTile/caseLandingSection/caseLandingForm/submitButton.

• **Domain-alias -** identification path within the framework to locate a defined object, or domain entity

https://localhost:8080/cmproduct/client/org/app/p/viewCMLandingRoot/cmLandingPage/cmLandingTile/caseLandingSection/caseLandingForm/submitButton.

• **Action -** instruction used to identify the operation to perform on the object identified by domainroot.

Ex: @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/tile/section/form

**• Query Parameters -** A standard set of query parameters

@Config(url=“/p/viewCMLandingRoot/cmLandingPage/cmLandingTile/caseLandingSection/caseLandingForm/\_process?fn=\_set&url=/p/employee/\_search?fn=query&where=employee.id.eq(<!/../.m/id!>)&orderby= employee.lastModifiedDate.desc()"),

In the above config fn is one of the reserved keyword and \_query is one of the predefined function Handler., \_search is the Action

--

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

**The possible Actions are: -**

**Action :**

/\* CRUD \*/

\_get, //HTTP GET - defaults to \_detail

\_save, //HTTP GET

\_new, //HTTP POST

\_replace, //HTTP PUT - full update

\_update, //HTTP PATCH- partial update

\_delete, //HTTP DELETE

/\* transient state \*/

\_search,

\_config,

/\* process \*/

\_process, //Allows for custom process/work-flow definitions

/\* navigation \*/

\_nav

Note: Action DEFAULT = \_get;

**Action:**

• \_new: Creates a new instance for the model

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

• \_replace: Replaces the model state

• \_update: Updates the model state

• \_remove: Removes the model from the database

• \_search: Searches the model based on a search criteria

• \_process: Executes asigned workflow process or custom handlers

**Link for technical info for predefined Actions:**

https://github.com/openanthem/nimbus-core/blob/master/nimbus-core/src/main/java/com/antheminc/oss/nimbus/app/extension/config/DefaultCoreExecutorConfig.java

--

Function Handlers are an abstraction within the framework to execute/handle a common set of f/w instructions for a given Action.

**Predefined Function Handlers**

There are several default function handlers defined within the core framework to handle common framework instructions, such as setting the state of a parameter (set) or adding a parameter value into a collection (add).

**Predefined Function Handlers**

**Handler Action** **Description**

\_param \_get

\_default \_nav

\_initEntity \_new

\_add \_process Adds parameter value to a collection.

\_addCollection \_process

\_bpm \_process Invokes a statless bpm process.

\_eval \_process

\_set \_process Sets parameter value.

\_setByRule \_process Set parameter value through a rule.

\_example \_search

\_lookup \_search

\_query \_search

**Using function handlers with respect to particular action:**

\_new?fn=\_initEntity

\_get?fn=param

1) \_nav?fn=default or 2) \_nav?key = , example: \_nav?pageId=

\_process?fn=\_set

\_process?fn=\_add

\_process?fn=\_update

\_process?fn=\_setByRule

\_process?fn=\_bpm

\_search?fn=lookup

\_search?fn=example

\_search?fn=query

\_process?fn=\_eval

--

**View Config Annotations:**

Once check mentioned basic view config annotations and its sample code in reference document.

1)@Textbox 2)@Textarea 3)@calender, 4)@Radio 5)@ButtonGroup 6)@Button 7)@Grid.. etc

--

**Code Conventions:**

**1)Annotation Ordering**

Consider organizing fields decorated with framework config annotations to a standardized ordering.

**The following order is suggested:**

• Label Annotation

• Constraint Annotations

• View Config Annotation

• Param Config Annotations

• Conditional Annotations

• Config Annotations

Why? Maintainability and Readability.

Ex:

@Label("Notification Preference")

@NotNull

@ComboBox(postEventOnChange = true)

@Path

@Values(OwnerNotificationPreferences.class)

@ValidateConditional(when = "state == 'physical\_mail'", scope = ValidationScope.CHILDREN, targetPath = "../contactInfo", targetGroup = GROUP\_1.class)

@ValidateConditional(when = "state == 'email'", scope = ValidationScope.CHILDREN, targetPath = "../contactInfo", targetGroup = GROUP\_3.class)

@Config(url = "<!#this!>/../initialized/\_process?fn=\_set&value=true")

private String notificationPreference;

**Reserved Keywords:**

*There are many reserved keywords used in framework mostly used while using @Config i.e ()*

**Ex** *Command Query Dsl Query***:**

@Config(url=“/p/viewCMLandingRoot/cmLandingPage/cmLandingTile/caseLandingSection/caseLandingForm/\_process?fn=\_set&url=/p/employee/\_search?fn=query&where=employee.id.eq(<!/.d/.m/id!>)&orderby= employee.lastModifiedDate.desc()")

**Reserved Keywords** Like

1).d 2).m 3)<! ! > 4) <! #this! > 5) <!#self ! > 6)$ 7) & 8)/p 9)json 10)rawpayload 11) <!#env!> 12)/../ 13)& … etc

Framework parse this reserved keywords in the url of the @config to execute it .

**Link for the Parser used by the frame work:**

<https://github.com/openanthem/nimbus-core/blob/master/nimbus-core/src/main/java/com/antheminc/oss/nimbus/domain/cmd/exec/internal/DefaultCommandPathVariableResolver.java>

Having knowledge on these reserved keywords will be helpful while implemented **@Config/@Configs:**

/../ (used to specify the path/switch from component to other)

/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.

<! #this! > Resolves to the path of the param in the current context.

<! #self! > When preceeded with loginId or id, resolves to the respective field fom the current logged in user’s ClientUser object. Otherwise, resolves to the current Client Alias.

<!#refId!> Resolves to the RefId of the ExecutionContext from which the command is being executed.

<!#env!> Uses the Spring Environment to resolve the preceeding path property. (e.g. <!#env.spring.application.name!> resolves using Environment.getProperty("spring.application.name"))

….. etc

For all reserved keywords, once see the basic contants , nimbus frame work understands:

**Link for all the constants:**

<https://github.com/openanthem/nimbus-core/blob/master/nimbus-core/src/main/java/com/antheminc/oss/nimbus/domain/defn/Constants.java>

--- basic configs useful for crud:

**CRUD:**

1)save:

//For Insert the form details into the database(Insert as new record)

@Configs({

@Config(url = "/p/employee/\_new")

})

@Button(type = Button.Type.Submit)

Private String Submit

(or) to insert the form details into the database(Insert as new record) and redirecting to other page at a time(by using \_nav(navigate function handler))

@Configs({

@Config(url = "/p/employee /\_new"),

@Config(url="/p/employeeviewroot/\_nav?pageId=vpOwners")

})

2) retrive(config to load records in grid):

@Config(url="<!#this!>/.m/\_process?fn=\_set&url=/p/ employee /\_search?fn=query&where="+

"{\n" +

"\"aggregate\" : \" employee \" ,\n" +

"\"pipeline\" :\n" +

" [\n" +

" {\n" +

" \"$match\": {\n" +

" }\n" +

" },\n" +

" {\n" +

" \"$project\": {\n" +

" firstName:1,\n" +

" addressLine1:1,\n" +

" city:1,\n" +

" state:1,\n" +

" recipientEmail:1,\n" +

" faxNumber:1,\n" +

" phoneNumber:1,\n" +

" \"\_class\": \"com.anthem.cm.ltss.extension.model.cueintegration.core.Employee \"\n"+

" }\n" +

" }\n" +

"]}")

--

**general sample crud code:**

**Basics:**

1) //Example For instantiate and insert any core domain

@Config(url = "/p/employee/\_new"),

**CRUD:**

1)save:

//For Insert the form details into the database(Insert as new record)

@Configs({

@Config(url = "/p/employee/\_new"),

})

//For Insert the form details into the database(Insert as new record) and navigating to ther page after submit

(or)

@Configs({ @Config(url = "/p/employee/\_new"),

@Config(url="/p/ownerlandingview/\_nav?pageId=vpOwners")

})

2)Delete and edit links(in grid for each row):

@LinkMenu

private VLMOtherItemLinks vlmOtherItemLinks;

@Model

@Getter @Setter

public static class VLMOtherItemLinks {

@Configs({@Config(url = "/p/employee:<!/../../id!>/\_delete"),

@Config(url = "<!#this!>/../../\_delete")

})

@Link()

@Label(value = "Delete")

private String delete;

//For Edit values from grid and insert into the Form

@Config(url = "/vpGenerateCorrespondence/vtGenerateCorrespondence/vmDeliveryModal/deliveryInfoModalBody/vfRecipientsOtherForm/vaOther/vfNewContactForm/.m/\_process?fn=\_set&url=/p/employee/\_search?fn=query&where=employee.id.eq(<!/../../.m/id!>)&fetch=1")

@Link()

@Label(value = "Edit")

private String edit;

}

3)update

@Configs({

@Config(url="/vpAddEditOwner/vtAddEditOwner/vsAddEditOwner/vfAddEditOwner/\_update"),

@Config(url="/vpAddEditOwner/vtAddEditOwner/vsAddEditOwner/vfAddEditOwner/\_nav?pageId=vpOwnerInfo")

})

(or)

@Configs({ @Config(url = "/vpGenerateCorrespondence/vtGenerateCorrespondence/vmDeliveryModal/deliveryInfoModalBody/vfRecipientsOtherForm/vaOther/vfNewContactForm/\_replace"),

@Config(url = "/p/employee:<!/../../id!>/\_replace?rawPayload=<!json(/../../)!>"),

@Config(url="/vpGenerateCorrespondence/vtGenerateCorrespondence/vmDeliveryModal/deliveryInfoModalBody/vfRecipientsOtherForm/vaOther/OtherRecipients/\_get")

})

4)retrieve(config to load records in grid):