# Fundamentals Introduction

## Guidewire product architecture

* Guidewire builds software products that help property and casualty (P&C) insurers replace their legacy core systems and transform their business.

### Product Landscape:

Set of applications

Monitoring &

Guidance

External Access

Mobile& GW data hub to store data (CM)& GW live n/w

Portal Guidewire Info Center is a BI warehouse

### Insurance Suite:

1. **Policy Center:** This application is designed to issue, modify and maintain data about policies
2. **Claim Center:** To manage the process of reporting, verifying and making payments on claims
3. **Billing Center:** To track and mange policy premiums and account changes
4. Depending on the configuration, Insurance Suite can contain other modules: Contact Manager, Client Data Management, Policy Center Rating, Policy Center Reinsurance, and Standards-Based templates.
5. Each application can function by itself, and each can be integrated with the other applications.

## GW configuration Technology

* Follows a Three-tier architecture, client-server architecture

### GW Application Tier:

* GW applications run within an application server basically java EE.
* Each Guidewire application is built and commonly deployed as a Web Application Archive (WAR) or Enterprise Application Archive (EAR) file to the application server
* This WAR or EAR file contain all the configuration, operational data and data definition files necessary to execute the application

### Data Tier:

* Data tier contains the business and operational database stores data in Relational DB
* GW supports Oracle SQL, Microsoft SQL, H2(for development) DBMS.
* war file contains a file ‘config.xml’ that specifies which database to connect to and how to establish the connection. DB is hosted in machine other than application hosting machine.

### Presentation Tier:

* End-users connect to the application using a web browser. **Chrome is the preferred browser**.
* Each application generates a collection of standard HTML pages that are rendered by the browser.
* Each application is a server-side application. It dynamically generates the HTML pages and data within those pages.
* It makes only a minimal use of functionality in the client (the web browser).
* Because there are no static HTML files, you cannot use the browser's Back button. All navigation is controlled on the server side.

## GW Platform:

* GW offers technologies for configuring

1. Data Model -> Entities
2. User Interface -> PCF
3. Application Logic -> Gosu Classes
4. Integration Points -> Gosu classes

### Important commands:

* To start the application: gwXX studio(run from …<Application>\bin)
* To generate documentation: gwXX regen-gosudoc (saved in …<Application>\build\gosudoc\)
* To generate data dictionary: gwXX regen-dictionary (saved in …<Application>\build\dictionary\)

### Navigation keystrokes

#### From studio

1. CTRL+N -> Search for a word with in a class
2. CTRL+SHIFT+N -> Search for a file
3. CTRL+SHIFT+F -> Search for a word
4. CTRL+ALT+SHIFT+N -> Search for a symbol
5. ALT+SHIFT+F7 -> Search for usages of function
6. ALT+SHIFT+F9 -> Run the server in Debug mode
7. ALT+SHIFT+F10 -> Run the server in Development mode

#### From UI (Google Chrome):

1. ALT+SHIFT+L -> Reload PCF’s
2. ALT+SHIFT+I -> Location info
3. ALT+SHIFT+W -> Widget Inspector

### Accessing the application:

* URL syntax to access a Guidewire application -> **<http://hostName:port/appCode>**

|  |  |  |
| --- | --- | --- |
| **APPLICATION NAME** | **PORT NUMBER** | **APP CODE** |
| Contact Manager | 8880 | ab |
| Policy Center | 8180 | pc |
| Claim Center | 8080 | cc |
| Billing Center | 8580 | bc |

### Terminating the application:

* When you are working in development mode, you can also stop the Guidewire application by terminating the batch job. To do this, press CTRL + C and then respond to the "terminate batch job?" prompt with a "y" (for yes). This has the advantage of keeping the command prompt window open and retaining the history of previously executed commands. This is useful if you need to stop and later restart the application.
* However, in some cases, the server may not release the port number. When this occurs, you need to execute a gwXX dev-stop before you can restart the application. You should never terminate the batch job for an instance running in production mode.

# Data Model

## Introduction:

* Set of data objects and information about their relationships
* Consists of entities, entity fields, typelists and typekeys.

## Entity:

* Entity is an abstract definition of group or business objects used by the application
* Data is stored in a database table

## Typelist:

* Each typelist has its data stored in a table
* Typekey is a foreign key to a single row in a typelist table

### Subtype typelists:

* Whenever an entity is subtyped, a typelist is automatically created for it.
* This typelist contains one type code for every subtype of the parent entity

# Entity in Detail:

* Activity and User are the entities that are common to all GW applications, so called as Platform specific entities
* All ETI and EIX(GW extended entities) which are found in metadata/entity are read-only files
* EIX files are extended platform specific entities.
* ETI files can also be created by customer in extensions/entity folder
* ETX files are the entity extensions in which customer can edit/create based on their requirements and found in extensions/entity folder

## Types of Entities:

1. **Virtual entities**: These are created during run time (using code) and not stored in DB so called as Non – persistent data
2. **Subtype entities**: These are children entities of a top – level entity(supertype entity) and shares the same database table of supertype which adds one more column(subtype) as a typkey which points to the subtype name and this table contains all the supertype entity columns and also subtype columns adding to them
3. **View entities:** These are entities used to improve performance for list view pages in rendering the UI
4. **Delegate entity:** A delegate is a virtual entity consisting of database fields, or methods, or both that can be reused by multiple entities

**Eg:**

1. **Assignable**
2. **EventAware**
3. **Validatable**
4. **Workitem**

## Properties:

1. **Abstract:** When this property set to true (Abstract entities)

* Super type entity is an abstract entity then These entities cannot be instantiated, but they can be subtyped and subtyped can create it own objects and access its own properties.
* Subtype entity can also be an abstract entity

**Note**: In Claim Center -> Contact, Incident and Transaction are super type entities and Contact and Transaction are Abstract entities

1. **Platform:** If this is specified as true Then that entity is a platform specific entity
2. **Final:** If this is specified as true Then that entity cannot be subtyped
3. **Extendable:** If this is specified as true Then we can create an extension to that entity

## Validation of an Entity:

1. Entity is an XML file which is validated using xsd/metadata/datamodel.xsd.
2. The primary elements of the xml are:
3. Entity declaration <entity>
4. Entity Extension <extension>
5. Subtype declaration <subtype>
6. Subtype extension <extension>

## Creating an Entity:

1. Navigate to Extension/entity folder
2. Right click -> New -> Entity
3. Edit the columns and specify their attributes
4. Optionally regenerate data dictionary
5. Deploy the changes

## Extending an Entity:

1. Navigate to Extension/entity folder
2. Right click -> New -> Entity Extension -> Pick up the entity from dialog box
3. Edit the columns and specify their attributes
4. Optionally regenerate data dictionary
5. Deploy the changes

## Entity Fields:

1. **Data Field:** Basically, a column.

* Name attribute -> Lower limit is maximum 30
* Type -> Data type (varchar,decimal,bit etc)
* null ok is a required attribute which need to be set manually since it has no default value
* <columnparam> is used to specify parameters based on type attribute of column

1. **Foreign key:**

* Pointer to single instance of some other entity column in database
* Stores a reference to a related object (uses Id to store in DB)

1. **Array key:**

* Collection of pointers to instances of some other entity that is maintained by code during runtime
* Stores a set of references to a set of related objects (Not stored in DB)
* Eg: ABContact entity has an array key contact Notes and contact Note entity should have a foreign key to ABContact.
* Referencing entity should contain a foreign key to parent table

1. **Type key:** A field associated with a specific type list. Stores a reference to typelist value(uses id of value)

## Entity Names:

* Defined in config/Entity Names
* This is a display name field that does not appear on data dictionary
* Reasons for creating entity name:

1. an object as a whole is displayed for example in dropdown that lists Claims.
2. an object's DisplayName is explicitly referenced for example, in an info bar widget with its value property set to Assignable Queue Name.( name – group name)

## Relationships in an entity:

1. **<onetoone>**

* <onetoone> element defines a single-valued association to another entity that has a one-to-one cardinality
* Provides a reverse pointer to an entity that is pointing at the <onetoone> entity through a foreign key
* A one-to-one relationship often splits a logical entity across multiple physical entities
* Purpose: To store more number of columns (sometimes cause more null values and more irrelevant data) in a single entity might be hectic. To avoid this, we go for one to one relationship

**Example:**

ABContact <onetoone> FinancialSummary

FinancialSummary<foreignkey>ABContact 

1. **<manytomany>**

* To add a many-to-many relationship between entity types to the data model, first create a separate versionable entity that represents the many-to-many relationship.
* Next, create foreign keys to each entity in the many-to-many relationship
* Then add a unique index consisting of the foreign keys in the relationship.
* **Example**:****

**3. Circular relationship:**

* Sometimes an entity need to refer itself. This can happen for non – subtyped and subtyped entities also
* GW does not allow an entity to refer to itself or two or more reference entities through a cyclic relationship
* GW recommends <edgeforeignkey> to achieve this type of behavior
* An edge foreign key from A to B introduces a new, hidden entity that has a foreign key to A and a foreign key to B. However, it does not create any direct foreign key from A to B. This ensures that the relationship information does not reference non-existent rows and safe ordering too.
* Example:

Group need to refer itself to obtain data about parentGroup. Here Group uses an <edgeForeignkey> with attribute fkentity as Group

# Subtypes in Detail:

* A subtype is an entity that is a child of supertype and inherits all the fields from super type
* Limit creation of subtype entity because a subtype can have only one parent and multiple inheritance is not supported. Subtype always inherits the parent entity which means deep level contains large number of columns in which there is a chance of most of the columns is not useful for subtype results in null values

## Creation of a subtype entity:

1. Navigate to Extensions/entity
2. Right click -> Create -> New -> Entity/Entity Extension (subtype – entity / subtype extension – entity extension)
3. Select subtype as type in the dialog box of entity creation
4. Specify the column and attributes
5. Deploy the changes

# Typelist in Detail:

* A typelist is a list of defined values
* When rendered in UI, typically appears as a dropdown list
* The typelists(tti/tix) that are found in Metadata/Typelist are platform based and base application typelist which are read only
* The typelists(tti/ttx) that are found in Extension/Typelist are custom typelist which are editable. New typelists can be created in this folder

## Kinds of Typelists:

1. Internal (tti/ tix)

* tix – Read only; GW extended typelists
* tti – Platform specific typelist – Read only has final attribute = true
* found in Metadata/typelist

1. Extendable: tti –

* Application specific typelist
* can be extended by end user
* found in Extensions/typelist

1. Custom: (tti/ttx) – End user can create/extend the typelists

* found in Extensions/typelist

## Creating a typelist/Extending a typelist

1. Create/Extend the typelist
2. Define the typecodes
3. Optionally regenerate data dictionary
4. Deploy the changes

## Typekey :

A typekey field is an entity field associate with the typelist

### Creation:

1. Add a typekey element in the entity/entity extension
2. Define the typelist attribute with the specified typelist

## Typelist filter:

A typelist filter define a subset of typecodes in the typelist

### Creation:

1. Add typefilter element in the typelist
2. Add include into Filter parameter and add typecodes values

### Referencing the typelist filter in an entity:

For typekey element in entity,Specify the typefilter attribute

## Typecode Category:

* A way to filter typelist values based on other typelist values
* Simple terms to configure dependent dropdowns,we use Category and Categorylist
* Income categories Tab in parent show child typecodes

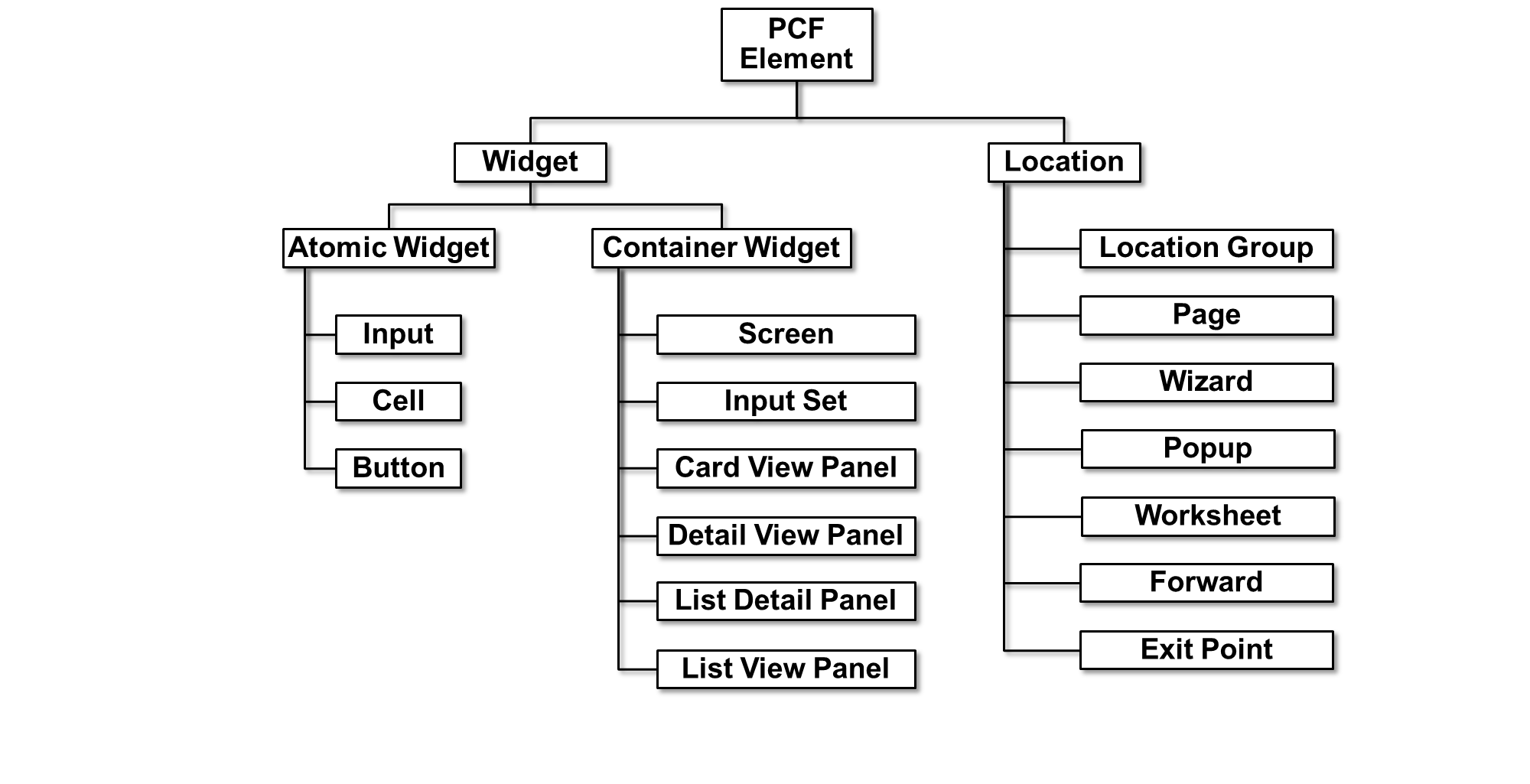
### Creation:

Add a Category parameter on the typecode and specify on which filter it should be displayed

# UI Architecture

## PCF element:

Page Configuration File object model is a proprietary application framework used to create all GW end user interface elements

Widget:

A PCF element that can be converted into HTML and displayed

#### Atomic widget:

An atomic piece of UI to display individual elements

#### Container widget:

A **container widget** is a collection of atomic widgets and/or other container widgets

##### Primary View:

Aprimary view is a reusable **panel** such as a detail view panel or list view panel that organizes atomic widgets.

##### Secondary View:

A **secondary view** organizes primary views. Card Detail Panel and List Detail Panel are used for this

##### Panel:

A **panel** is a container which has a structured UI layout, shares a common placement on a PCF page, and can be rendered with a Toolbar and/or a Title bar that controls the panel behavior.

Panels can be placed within screens using PanelRef widget.

Types of Panels: List View Panel, Detail View Panel, Card View Panel, List Detail Panel.

##### Screen:

a top-level container that one can navigate to

##### Input Set:

An **Input Set** is a reusable collection of atomic widgets for detail view panels

###### Steps to create a shared logic input set

1. Add an Input Set widget
2. Specify shared logic
3. Add atomic widgets
4. Deploy PCF

###### Steps to create a reusable input set

1. Create an input set PCF
2. Specify required variable(s)
3. Add atomic widgets
4. Reference the input set from parent
5. Deploy PCFs

##### Detail View Panel:

* A **detail view panel** is a container widget that allows user to view, and in some cases edit, data for one object and information related to that object.
* A detail view panel often has a root object and the atomic widgets in the DV are used to display the data elements of the root object
* A DV panel PCF file is a reusable file which ends with name DV
* A DV can also be created as a inline widget which means within a file and cannot be reused

###### Steps to create a DV panel PCF file:

1. Create the Detail View Panel PCF file
2. Specify the required variables
3. Optionally specify additional properties
4. Add input columns
5. Add input widgets
6. Deploy PCFs

###### Steps to reference a detail view panel

1. Add PanelRef widget
2. Specify widget properties
3. Deploy PCFs

##### List View Panel:

A **list view** **panel** is a container widget that often displays a set of rows that are related to one object or one query

###### Steps to create a List View PanelPCF

1. Create the List View Panel PCF file
2. Specify the required variables
3. Specify additional properties
4. Add row iterator to list view panel
5. Add row to row iterator
6. Add cell widgets to row
7. Deploy PCFs

###### Steps to reference a List View Panel

1. Add panel ref
2. Specify def property
3. Add toolbar
4. Deploy PCFs

**PAGE, WIZARD, POPUP, WORKSHEET**

**SCREEN**

**CARD VIEW PANEL /LIST DETAIL PANEL**

secondary view

**DETAIL VIEW PANEL**

**LIST VIEW PANEL**

primary view

**INPUT SET**

button, input, cell,

**ATOMIC WIDGETS**

#### Card View Panel:

* A **card view** is a collection of cards, each with one or more views
* Cards appear in the UI as tabs in the screen area
* Card views are useful when you want to present the user with several sets of information about a single business object, but you want only one set visible at any given point in time

##### Steps to create a card view

1. Create the card view file
2. Specify the required variables
3. Optionally specify additional properties
4. Add one or more cards
5. For each card, specify the card properties
6. For each card, specify the contents

#### List Detail View Panel:

* A **listdetail view** consists of two views: a list view on top with one selected row and a card view on bottom that displays details about the selected row
* The **selection name** propertynames the object in the selected row of the list view that is displayed in detail in the card view

##### Steps to create a List Detail View:

1. Create the listdetail view file
2. Specify the required variables
3. Specify the selection name and type
4. Add or reference a list view
5. Add or reference a card view

##### Input Column

Input columns organize layout and input widgets

###### Widget Properties:

Static Properties:

Evaluates a static value that is immutable, never changes

Id requires a static value.

Dynamic Properties:

After a user navigates to a page or clicks update, evaluates an expression and returns a value

Editable – Requires a Boolean expression/value

Label – Requires a String value (A value from display.properties file)

Required – Requires a Boolean expression/value

Value – A data value that needs to be displayed in UI

Available – Requires a Boolean expression/value

Visible – Requires a Boolean expression/value

#### Mode:

1. **Mode** is a property used for PCFs that are appropriate for a given business scenario
2. Modal PCF sets include one PCF with mode of "default", which is used when something   
   references a mode that doesn't otherwise exist
3. Modes can be used with:
4. Detail view panels
5. Card view panels
6. Input sets
7. List view panels
8. Info bars
9. Screens
10. Menu item sets

##### Steps to implement modal PCF’s:

1. Create modal PCF set
   1. For each PCF, specify its mode(s)
   2. Ensure there is one PCF with "default" mode
   3. Ensure each PCF has identical required variables (in number, type, and order)
2. Create reference to modal PCF set
3. Deploy PCFs

### Location:

Navigable places in application that a user or the application itself can navigate to

#### Page:

* A page contains a single screen

#### Location Group:

* A location group is a collection of pages used to view or modify the data
* Each group has single info bar, actions menu, and side bar

#### Wizard:

* A wizard is an ordered collection of screens used to execute a complex business process
* Single info bar, actions menu, and side bar
* Includes toolbar with Back and Next buttons

#### Popup:

* A **popup** contains a single screen and returns the user to the previous location once the popup is closed
* Automatically has "Return to <previous location>" link

##### Steps to create a popup:

1. Create a popup PCF file
2. Specify variable(s)
3. Specify entry point(s)
4. Specify properties
5. Add containers and input widgets
6. Configure the navigation widget
7. Deploy PCFs

#### Work Sheet:

* A **worksheet** contains a single screen rendered in the workspace frame
* Worksheets can appear and disappear (without explicitly being dismissed by the user) as the user moves from one top-level tab or object to another.
* A worksheet can also have a global scope which makes it always visible.

###### Steps to create a worksheet:

1. Create the worksheet file
2. Specify the required properties
   1. Basic properties -> id, title, tabLabel etc….
   2. Entry points
   3. Scope -> When you specify a scope for a worksheet, then the worksheet is visible only when the named location and named object are in the screen area. When the user navigates away from that location (or from that object within the same location), then the worksheet will no longer be visible.

If you want the scope of the worksheet to be global (meaning it will always be visible regardless of the location or object displayed in the screen area), then specify "\*" for the scope.

* 1. Variables -> There must be one variable for each root object of the location. We need to specify the initial value of the variable

1. Optionally specify additional properties -> Parent property - This identifies where in the UI hierarchy the worksheet exists. This is used in the Unsaved Work menu. (All unsaved work for a given location is listed under a link for that location.)
2. Add a screen to the worksheet
3. Add or reference a detail view
4. Create a navigation widget

#### Forward:

* A **forward** contains logic to execute before navigating to another location
* Often involves deciding which location to navigate to

##### Steps to create a forward:

1. Create the forward file
2. Specify any required properties
3. Add one or more forward conditions
4. Set forward condition properties
5. Ensure final forward condition always executes if needed (otherwise it gives an error in UI)
6. Create navigation widget

#### Exit Point:

* An **exit point** points to a URL outside of the Guidewire application
* Often used to access other applications or websites
* Does not contain (directly or indirectly) a screen widget

##### Steps to implement exit point:

1. Create the exit point
2. Specify the exit point properties, including:
   * The external destination URL
   * Properties for the window in which the destination will be opened
3. Configure a widget to navigate to the exit point

To navigate to a location from one screen, you need to specify the action with id name of the location and function **go()** which required variables are passed as a parameter to the location.

# Advanced UI Architecture:

## Entity Names:

* An **entity name** defines how to display a name for an instance of a given entity
* An entity name file contains at least one default entity name type definition which can be accessible through DisplayName property.
* **Usage:**

1. When displaying an entity instance Eg: Dropdown list
2. When explicitly referencing the DisplayName property of an entity instance

### Steps to create an entity name type

1. Create a new entity name file(.en file)
2. Define variables in the variable table
3. Define a return value expression for the default entity name type
4. [Optional] Define custom entity name types
5. Deploy entity name types

## Dynamic Dropdowns:

* A **dynamic dropdown** is a list of values populated from run-time data
* Values typically come from an array, but can come from any list
* Implemented with object sets and range widgets
* Typically, independent of any other single field

### Range Widgets:

Dynamic dropdowns are created using

1. Range input widget (for DV)
2. Range Cell Widget (for LV)

**Value Range:** Used to specify list of values

**Value:** to store selected value

### Step to create dynamic dropdown

1. Determine the array to use
2. Create the range widget
3. Specify the range widget's properties

## Dependent Dropdowns:

* Dependent drop-down lists associate parent and child
* Parent selection dynamically filters child list values
* In an Entity, Typekey keyfilters specify the dependency relationship
* Typelist category and categorylist filters specify the filtering

### Steps to create dependent drop-down lists

1. Add category filter and/or categorylist filter to child typelist
2. Configure typekey keyfilter
3. Add input widgets to PCF
4. Deploy your changes

### Difference between type filter and key filter:

**TypeFilter:** You define a typefilter in a typelist. A typefilter specifies a static (fixed) set of typecode values from the typelist itself. The typecodes can be included or excluded from a typefilter. An entity that contains a typekey for a typelist with a typefilter can reference the typefilter by name in the typefilter attribute

**KeyFilter:** You define a keyfilter for a specific typekey in an entity. The keyfilter specifies the parent typekey in the same entity definition that filters the child typekey values. The configuration of the child typelist determines how the typecodes are filtered by the parent typelist.

## Iterators:

* Within the PCF architecture, an **iterator** is a PCF element that takes a collection of values and creates one UI element of a given type for each value in the collection
* **elementName** property is an arbitrary name used in the child PCF element to reference each item in the collection of values
* The PCF architecture includes the following iterators:
  + Row iterator
  + Input iterator
  + Cell iterator
  + Link iterator
  + Card iterator
  + Panel iterator
  + Menu item iterator

### Steps to create an iterator

1. Create the iterator
2. Specify the iterator properties
3. Add a child PCF element of the appropriate type
4. Configure the child element as needed

## Partial Page Update:

* Partial page update configuration required for dynamic widget behavior
* Dynamic widget behavior means - While a user changes business data, the widget property expression evaluated. This applies to all dynamic properties of a widget.

### Partial page update: layout re-ender

* Responds to a user changing business data **while** it happens and changes the layout of widgets on the screen
* Applies to following dynamic widget properties

1. Visibility
2. Editability
3. Availability
4. Required

### Partial page update: DATA\_ONLY

* Data is updated for the page
* No data committed
* Applies to only one dynamic widget property : Value

### Partial page update is PostOnChange

* Configurable widgets
  + Cell
  + Input
  + InputGroup
* PostOnChange tab property
* Mark the Enable targeted Post On Change checkbox
* Three (3) configurable properties
  + disablePostOnEnter, onChange, target

#### Property: disablePostOnEnter

* Default is false
* If evaluated to true, **not** triggered when page is rendered

#### Property: onChange

* Defines the Gosu expression to invoke
* Applicable when user changes the editable value of a widget
* Triggers changes to other widgets prior to the value being saved
* <PostOnChange/> must be present for onChange to trigger

#### Property: target

##### widget ID

* Target widget re-rendered
* Refreshes all user-editable page data

##### DATA\_ONLY

* Refreshes all user-editable page data
  + Input data
  + ListViewPanel row data
* NO layout updates
* Best performing

## Field level Validation:

* **Field-level validation** is a validation behavior tied to one or more specific data fields

### UI field level validation -> Configure widget properties:

1. Regex:

* **After** a user navigates to a page or clicks Update, the widget property expression is evaluated
* Shows value invalid field warning
* Shows info bar field error
* Restricts data commit

1. inputMask:

* **After** a user navigates to a page or clicks Update, the widget property expression is evaluated
* Field watermark
* Placeholder causes implicit regular expression and shows field format warning
* NO data commit restriction

1. Validation Expression:

* **Before** commit, expression evaluated
* Shows field error
* Shows info bar message error
* Restricts data commit

### Data Model Field level validation:

#### Data types validation:

* All field values in the UI are displayed and entered as strings
* Data types ensure that field values are valid types that can be saved to the database
* Data types are API enforced

#### Field validator:

* A **field validator** defines the pattern for valid field values
* Can be associated with one or more fields and one or more entities
* If the value does not match the pattern, an error message is displayed, and data is not commit

##### Steps to implement field validator:

1. Create a validator display key error message
2. Create a field validator
3. Associate the field validator with an entity element
4. Deploy your changes

Eg: <ValidatorDef desc = “” input-mask= “” name=”” value=””/>

### UI notifications:

* **gw.api.util.LocationUtil** contains static methods
  + **addRequestScopeInfo(*String*)** to show information message
  + **addRequestScopeWarning(*String*)** to show warning message
  + **addRequestScopeError(*String*)** to show error message
* **gw.api.util.DisplayableException –** uses throw new keywords to display error message