Technical Architecture (one document version)

GBST FrontOffice Technical Architecture

Authors:
Thomas Buckel
Terence Mackie
Greg Tan

This documentation is for GBST FrontOffice v0.0.7.

Table of Contents:

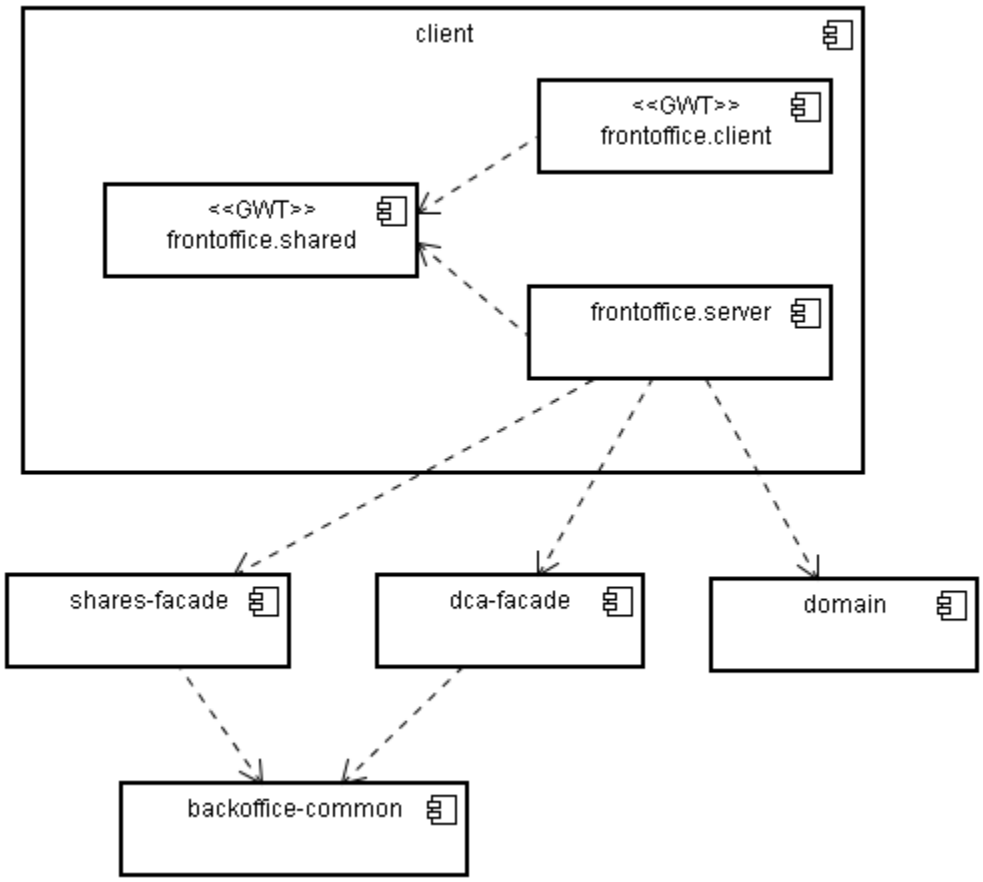
FrontOffice Technical Architecture

This document describes key architectural concepts of the GBST FrontOffice application. The intention is to give an overview of these concepts and how they are used in the FrontOffice application.

It is structure by the application's modules and also shows examplanatory the implementation of two use cases

(Showing Holdings and Confirmations).

Module Structure and Responsibilities



As shown in the above diagram, FrontOffice consists of these modules (which are implemented as Maven modules as well):

Module	packaging	Description
`client`	WAR	GWT module of the application containing this core structure: • client - Client side presentation code and logic following the Model-View-Presenter (MVP) pattern and using an Action framework • shared - Actions, Results and Data transfer objects (DTOs) used by RPC communication • server - Server side implementation of Actions by ActionHandlers that provision data from the FrontOffice domain, Shares and DCA.

`domain`	JAR	Domain model of the application. The domain classes are implemented as JPA entities and correspond to the tables in the FrontOffice database.
`backoffice-common`	JAR	Common classes used by the BackOffice facades.
`shares-facade`	JAR	Facade to Shares, exposing a SharesService interface and various Shares specific models. The Shares database is mapped through iBatis (a persistence framework which automates the mapping between SQL databases and objects in Java) into POJOs representing the results from either SQL queries against tables and views, or results from calling procedures.
`dca-facade`	JAR	Facade to DCA, similar to shares-facade, exposing a DCAService interface and various DCA specific models.
`database`	POM	Module containing the SQL scripts to create a SQL server database for the domain module. **Note:** This is subject to be removed and incorporated in the domain module and is subject to be reworked with Phase 1.

Used Technologies and Frameworks

- GWT 2.4 - Web UI and RPC communication
- GIN - Google Guice for GWT (Dependency Injection for GWT)
- Spring 2.5.6 - Server side framework for Dependency Injection, Transactions, JMS and JPA abstractions
- Spring Security - Authentication and Authorisation for web site and RPC calls
- JPA / Eclipselink - Server side persistence of the FrontOffice domain model
- iBatis - JDBC framework used for Shares and DCA access
- JasperReports - Report generation
- JUnit 4 - Unit testing
- Maven - Build system
- Dozer - POJO mapping

UML diagrams

UML diagrams on this page are created using astah Community (former Jude/Community) and can be found in the

project's svn repository in `/docs/GBST Front Office Model.jude`

Module 'domain'

The domain model implementation uses JPA for OR mapping and is implemented in the `domain` module. The entities are persisted in a MS SQL Server database.
The entities support optimistic locking through a version column supported by JPA.

Package Structure

domain module package structure

```
com.gbst.frontoffice.domain
- dao                          DAOs for accessing the domain model
- model                        All Model entities are in this package or subpackages
of it.
                               Core entities are in this package.
  - accounts                   Account entities
  - alerts                     Alert related entities
  - roles                      PartyRole entities
    - relationships            PartyRoleRelationship entities
- service                      Subject to removal: Contains an AlertService for
setting correct transactional demarcation.
```

Key entities

The FrontOffice domain model is built around Parties, PartyRoles, Accounts and relationships between these entities.

Party

A party can be a natural person, a legal entity or a group consisting to other Parties. Each Party can have multiple PartyRoles assigned defining the role(s) of a Party within the FrontOffice application.
Each unique person or legal entity will be represented by the same instance in any context of the application. Parties do not have any relationships to other Parties, compositions of Parties are defined by PartyRoles and the relationships of PartyRoles between themselves.
Parties can have addresses and contact details.

PartyRole

Depending on the nature of a role, it can have relationships to other accounts as well as relationships to other PartyRoles.
Relationships between roles are modelled through PartyRoleRelationships of various types (subclasses in the object model). Relationships to accounts are modelled by PartyRoleAccount relationships of various types.

Account

An abstract entity where a subclass for each integrated BackOffice system (Shares and DCA) exists, potentially holding system specific information. Different account types within the systems are not modelled with the FrontOffice

domain (yet).

Advisor Codes

Each account is linked to Advisor code(s) in its BackOffice system. The access to accounts is restricted by the advisor codes a user (Advisor or Assistant) has access to. The advisor codes an advisor or assistant has access to is modelled by an abstract role 'AdviserCodeRole' which links advisor codes to the AdviserRole or AdministratorRole.

Refer to "Account Security" for further details.

Examples

Person with one account

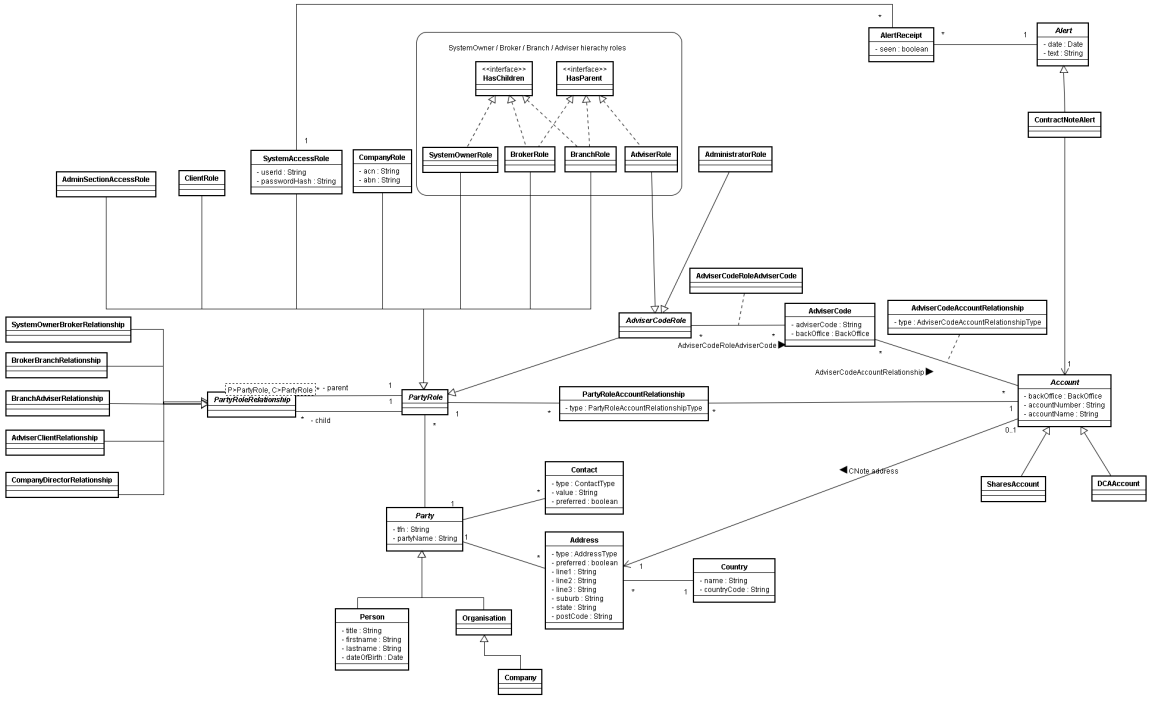
A party of type Person has a role 'ClientRole'. The ClientRole has a relationship to the owned Account.

Company with 2 directors

For each director a party of type Person exists. Each of this persons has their own DirectorRole. Another Party of type Company exists having a company role. The relationship between the directors and company is built by linking the two DirectorRoles to the Company's CompanyRole.

Class Diagram

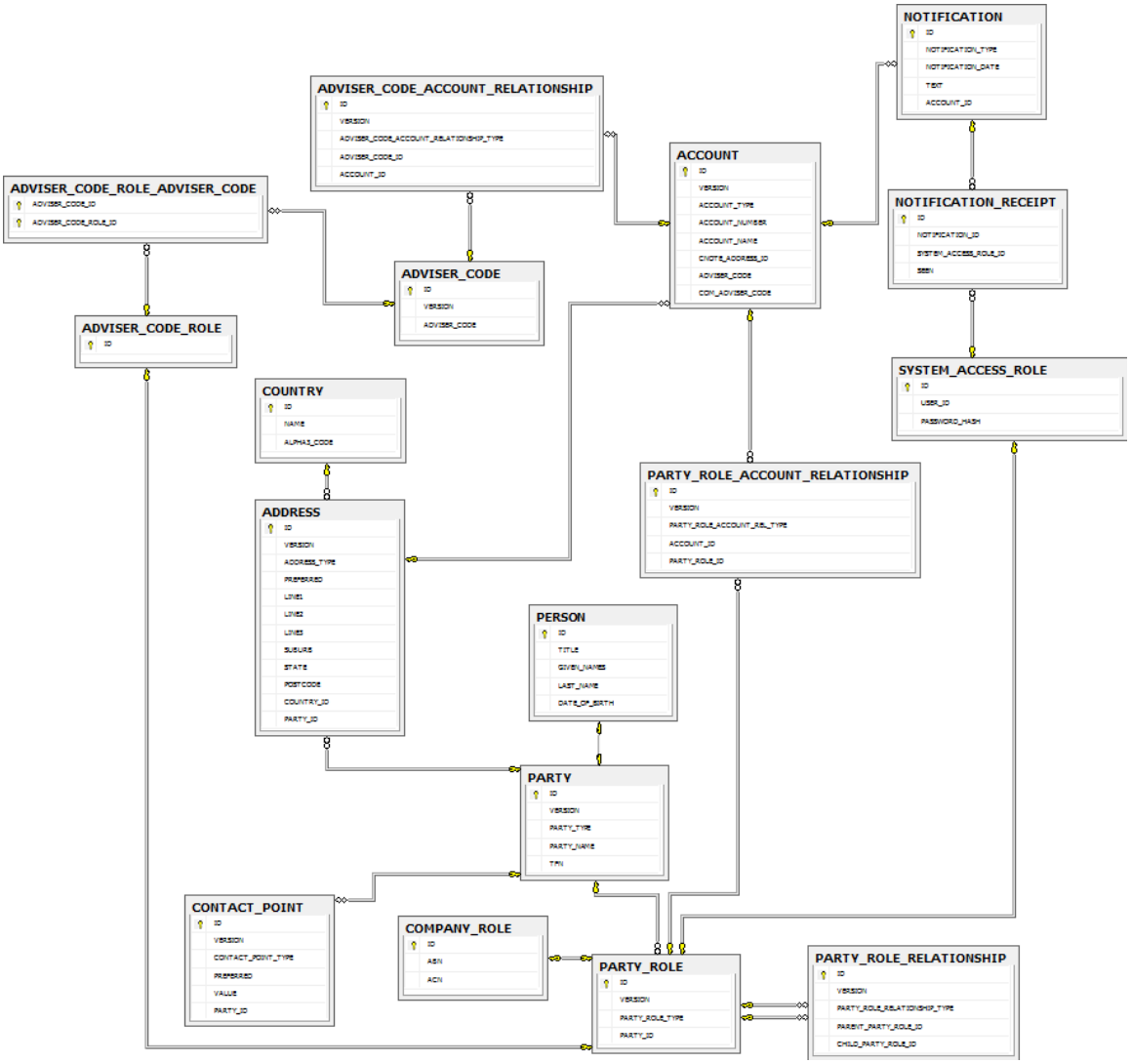
The following class diagram shows the application's domain model:



Entity Relationship Diagram

The domain model's entities are mapped to database tables through JPA. The applied strategy of class inheritance is the 'Joined' strategy, "A strategy in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass". For subclasses without any persistent fields, no tables are created.

The resulting ER Diagram for the FrontOffice database/schema:



Account Security

As briefly described in the domain model overview, FrontOffice currently uses 3 roles to determine whether a user has access to an account:

- Adviser & Assistant

Both Adviser and Assistant roles have explicitly assigned adviser codes. They can only access the accounts that are linked to these adviser codes in the backoffice system. Advisers and Assistants can only view clients which have at least one account with an advisor code in common with their assigned advisor codes.

- Administrator

The `AdminSectionAccessRole` grants a user access to the FrontOffice's Admin section. Within this section administrator can create clients, advisors and assistants. They can also assign access to Adviser Codes to advisers and assistants. An administrator can see all accounts.

Account security is implemented by restricting the accounts and clients returned to users without the `AdminSectionAccessRole` when searching. The `PartyDAO.findAccountsForSearch` and `PartyDAO.findClientsForSearch` methods apply account based access control to their database queries. Additional parts of the application

that display accounts and clients, such as the top clients widget, query their data based on the adviser codes that are linked to the logged in user. As the only two ways in which users can display account or client data is using the search, or clicking a link which sources account information based on linked adviser codes, the user will not be able to display data for which he does not have access.

Alerts which are linked to accounts use the `AccountSecurityDAO` to determine which users have access to the account when determining the recipients of the alert.

Explicit checks for account security are currently not performed on RPC calls to retrieve data. Implementing these checks will ensure that malicious users cannot use specially crafted RPC calls (not generated by the application) to obtain client or account data to which they do not have access. These checks will be implemented in a subsequent phase pending confirmation of the account security model.

At the moment all account information is accessible once a user can access an account.

Unit tests

Unit tests for the domain model use an embedded in-memory database (Apache Derby). This database is re-create via EclipseLink's generation option for EVERY test, i.e. every test runs on a clean database and has to set up its own fixtures if required.

Module 'client'

Package structure

The overall package structure of this module consists of three major parts:

Package name com.gbst.frontoffice.	compiled to Javascript	Description
client	yes	GWT client code • Presenter and Display implementations • GIN module definition • UI Widgets and utility classes
shared	yes	Shared (serialized) Objects between client and server: • Actions, Results and Callbacks • Data transfer objects (DTOs) • Streaming Messages • Exceptions

server	no	Server side classes running Spring environment: • ActionHandler that implements the server-side of Actions and use DAOs from domain module as well as the `SharesService` and `DCAService` • Jasper Reports integration • Databus connectivity • Scheduled jobs • Streaming

The GWT module is defined in `com.gbst.frontoffice.FrontOffice.gwt.xml`. The GWT EntryPoint of the application is the class `FrontOffice`. It delegates all initialisation to the `RootPresenter` which builds the Presenter/Display hierarchy and initialises the application.

Package structure of module 'client'

```
com.gbst.frontoffice
- client              - Definition of Action framework (client), GWT entry point,
RootPresenter and RootDisplay
  - alerts              - Presenter/Display for Alerts
  - events              - Events on the eventBus
  - history             - History mechanism
  - inject              - GIN module definition and providers
  - nav                 - Top and second level navigation (Presenter/Display)
  - report              - Presenter/Display for Report integration
  - search              - Advanced Search and QuickSearch
  - section             - Subpackages for each section containing the
Presenters/Displays
    - ..
  - service             - RPC service definitions
  - session             - Session handling
  - streaming           - Client side streaming implementation
  - util                - Utility classes
  - widgets             - GWT Widgets
    - composite           - extended Composite the fires events on
attachment/detachment to the document tree.
    - decorator           - Various DecoratorPanel subclasses.
    - forms               - Lightweight framework for forms and error messages in
MVP context
    - table               - Table abstraction on top of GWT incubator's ScrollTable
  - workspace           - Section and workspace management

- shared              - Serialized Exceptions
  - actions             - Actions, Results and Callbacks
    - ...                 - subpackages grouped by business functionality
  - dto                 - Data transfer objects and EntityIds
    - ...
  - streaming.message   - Streaming messages
  - util                - Utility classes, currently common date/time/numberformat
strings

- server              - Implementation of FrontOffice and Security services
  - alerts              - Expire Alerts Job
  - calcs.commission    - Common calculations shared by ActionHandlers and Report
datasources
  - databus             - Shares databus connectivity
    - cnotes              - Cnote message handling
  - handlers            - ActionHandler definition and ActionHandlerRegistry
    - ..                  - subpackges with ActionHandler implementations grouped by
business functionality
                          parallel to subpackages in shared.dto
  - jasperreports       - Report generation related classes. Contains a Servlet that
retrieves a generated report
    - datasource          - Datasource implementations for the various reports
  - market              - A MarketInformationProvider abstract with ASX website
implementation as well as
                        Index Market info publisher.
  - security            - Authentication and Authorisation related classes used with
Spring Security
  - spring              - GWT RPC Service Exporter that supports continuations
  - streaming           - Streaming implementation
```

Overview

The implementation of the GWT client/server is following current best practises, namely

- Model View Presenter pattern
- Command pattern for remote calls
- An EventBus on the client side to propagate application events

These concepts are explained in this presentation "Best Practices for Architecting GWT Apps" http://www.youtube.com/watch?v=PDuhR18-EdM

The following sections outline the implementation and usage of these concepts.

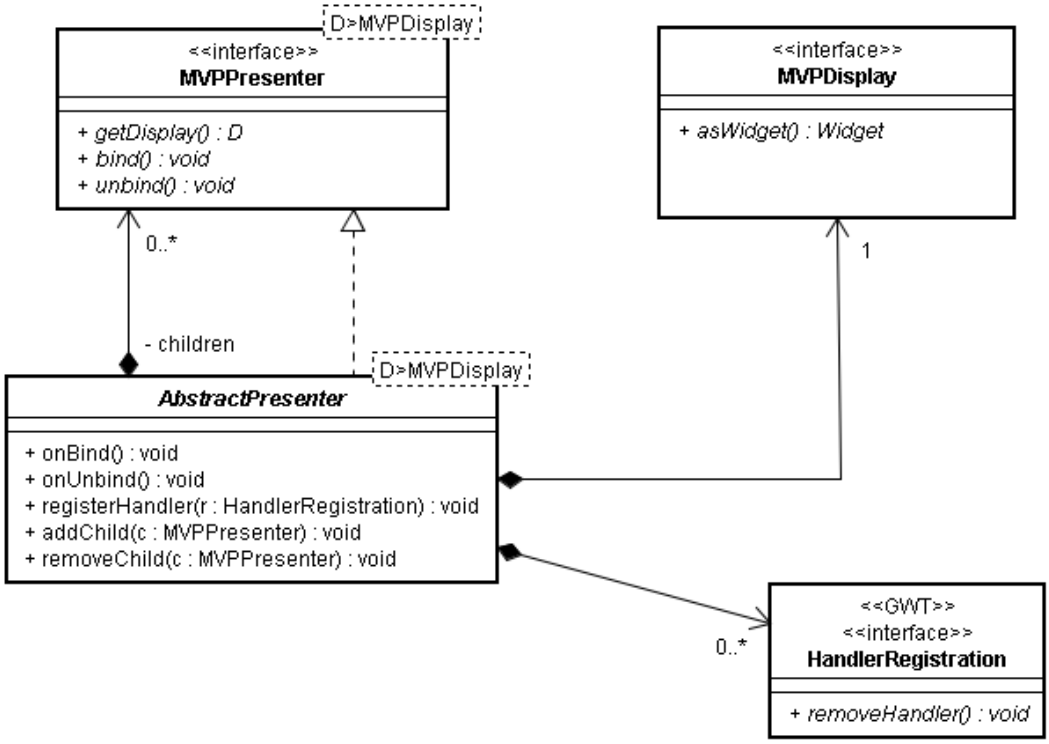
GWT client concepts

GIN

FrontOffice uses Google GIN (Guice for GWT) for dependency injection within the GWT client. The GIN bindings are defined in the classes `FrontOfficeModule` and `AdminModule`.
A so-called 'Ginjector' exposes only 2 classes for the entry point of the GWT application - the implementation of the RootPresenter and the eventBus, both are required for the initialisation of the application. All other instances are injected using GIN's constructor injection.
Custom providers (ie Factories) exist for classes that requires instantiation through `GWT.create()`, i.e. the RPC service stubs and GWT `Constant` implementations.

Model View Presenter

The implementation of the presentation tier follows the Model-View-Presenter pattern (the View is named Display though).



There are basic interfaces each Presenter (`MVPPresenter`) and Display (`MVPDisplay`) has to implement. `AbstractPresenter` provides an implementation of common Presenter logic.

- All logic is to be implemented in the Presenter, the Display just provides handles (interfaces like HasValue or HasClickHandlers) to UI components.
- The Presenter implementation defines the interface for its display. This interface has NO dependency on GWT Widgets, therefore a mock display can be injected into a Presenter outside the GWT/browser environment for quick unit testing.
- A Presenter's Display interface is usually an inner class of the Presenter implementation named Display - "The Presenter's implementation defines what information/elements it requires on the Display".
- Presenters reference only presenters, Displays reference only displays,
- In parallel to the MVP pattern, properly interfaced custom Widgets the contain no application logic (e.g. Table implementations) are used by the Displays.
- Presenter are bound and unbound. During 'bind' a Presenter has to register its required listeners to either Display elements or the Eventbus. During 'unbind' these listeners have to be unregistered.

`AbstractPresenter`

- Implements core functionality for Presenters:
  - Holds a reference to its Display
  - Holds a list of registered handlers that will automatically be unregistered on unbind if `registerHandler()` is used:

```
registerHandler(getDisplay().getSaveButton().addClickHandler(new
ClickHandler() {
    @Override
    public void onClick(ClickEvent event) {
        onSave();
    }
}));
```

- Support for child presenters that are automatically bound/unbound with the Presenter.
- It avoids double binding/unbinding.

FrontOffice follows this naming patterns for Presenters and Displays:

- If a Presenter requires additional interface methods to MVPPresenter, the interface is suffixed with Presenter and the implementation is suffixed with PresenterImpl. Otherwise the implementation is named Presenter extending `MVPPresenter`.
- Usually the Display interface is named Display and is defined as inner class/interface in the Presenter implementation and the implementation is suffixed with Display.

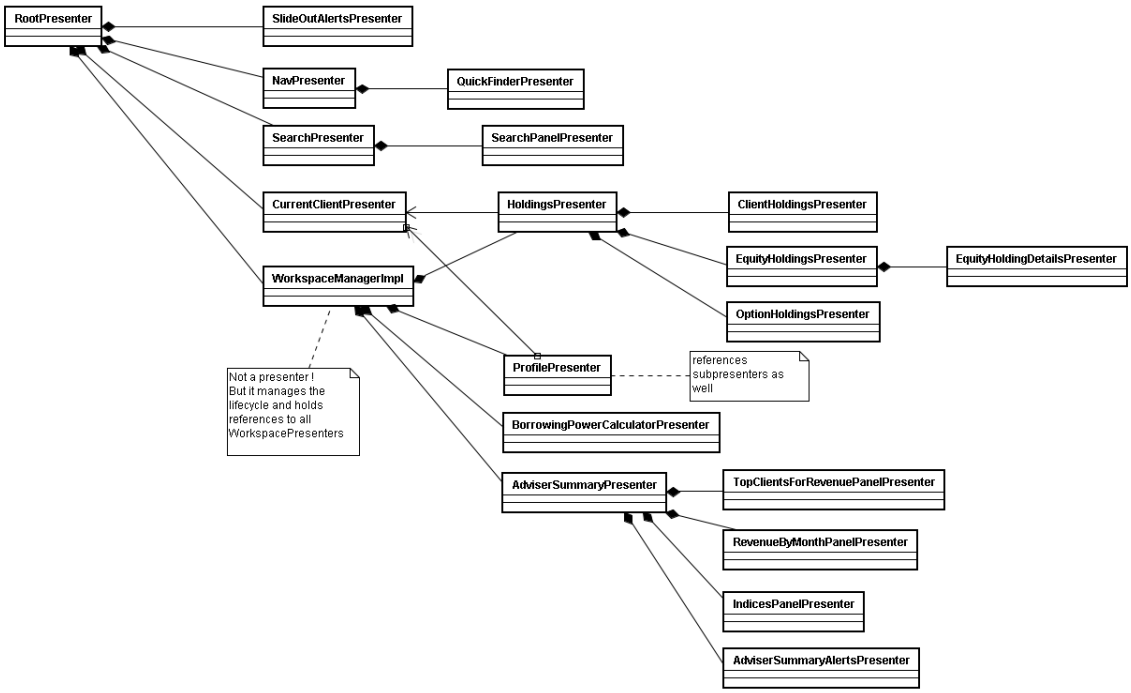
Example:

- BorrowingPowerCalculatorPresenter implements MVPPresenter
- BorrowingPowerCalculatorPresenter.Display implements MVPDisplay
- BorrowingPowerCalculatorDisplay implements BorrowingPowerCalculatorPresenter.Display

Presenter Hierarchy

The following diagram shows a representative view on the hierarchy of the presenters.
All references of the RootPresenter are shown, but only a selection of 4 representative Presenters from the WorkspaceSpaceManager:

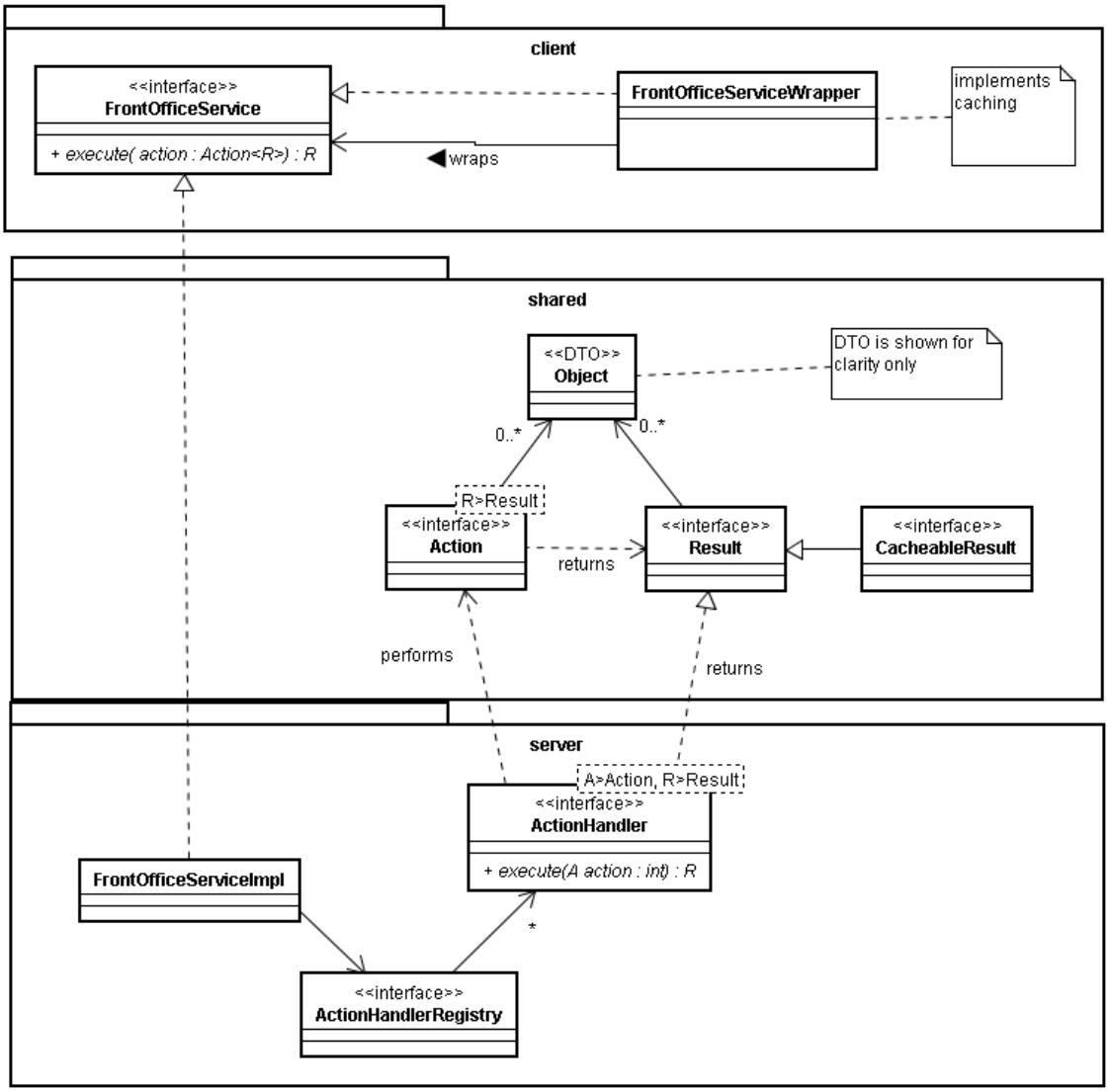
- HoldingsPresenter and ProfilePresenter are both implementing `AbstractClientSectionPresenter` and they reference the same instance of `CurrentClientPresenter`
- `AdviserSummaryPresenter` has four child presenters but is not depending on the currently selected client
- `BorrowingPowerCalculatorPresenter` is a simple WorkspacePresenter and has no child presenters



Note: The diagram shows all classes without the *Impl suffix for clarity. `WorkspaceManagerImpl` holds references to all `WorkspacePresenters`, only 4 of these are shown in above diagram.

Action Framework

Action, Result and ActionHandler



The Action framework implements the Command pattern built on top of GWT's RPC mechanism. Each remote call is implemented as its own Action (Command is a already defined class in GWT hence the name Action) which returns a Result. Both Action and Result can have (a) DTO(s) as payload. There is no common interface/super class for DTOs.

An ActionHandler executes the Action on the server side. Each ActionHandler defines the Action it executes. The binding information is provided by the ActionHandler itself (method `getActionType()`, allowing it to be replaced by mock implementations without changing the Action implementations or anything else.

FrontOfficeService is a standard GWT remote service (client interface, async interface, server side implementation) executing an Action and returning a Result. A typed AsyncCallback implementation is implemented for each concrete Action/Result pair extending GWT's AsyncCallbackInterface. The server implementation of this service looks up the ActionHandler for an Action and delegates processing to it.

The GWT client is using a wrapped implementation of the FrontOfficeService (inject and created through Spring) which centralises Exception handling (e.g. session timed out) and local caching of Results that implement the CacheableResult interface.

ActionHandlers

ActionHandlers define the transaction boundaries (usually through Spring's @Transactional annotation) and can be secured to certain roles by Spring Security (@Secured annotation).
All ActionHandlers are registered in an `ActionHandlerRegistry` using Spring's qualified component scan (`@Qualifier("actionHandler")` annotation).
The mapping between domain/backoffice entities and DTOs is done using Dozer while custom mapping implementations exist for more complex mapping. Dozer's configuration is defined in `dto-domain-mappings.xml`.

DTOs - Data Transfer Objects

All data exchanged between client and server as payload of Actions and Results is implemented a data transfer objects (`DTOs`) which are plain POJOs. The reason for not serialising entities from the domain model (besides potential lazy loading problems) is to have tailored DTOs for Actions/Results, minimising the number of remote calls as well as the amount of transferred data.

Within DTOs, every Id of entities of the domain model shall only be referenced by a subclass of `EntityId`. This allows a typed usage of Ids (rather than longs or similar). `EntityId` also holds a `version` originated from the optimistic locking in the domain model. In case of update actions, this version can be used within JPA to detected update conflicts.

For future implementation: Identifying values from Shares or DCA shall only be used in a typed way as well.

A note on GWT serialization:

Actions, Results and data transfer objects (DTOs) have to be serializable. As GWT serialization does not support final fields, the convention is to have a private default constructor (required only for GWT internally) and a constructor with the required fields of the class.

```
public class GetCountriesResult implements Result {

    private ArrayList<CountryDTO> countries;
    private CountryId defaultCountryId;

    @SuppressWarnings({"UnusedDeclaration"})
    private GetCountriesResult() {
    }

    public GetCountriesResult(ArrayList<CountryDTO> countries, CountryId
defaultCountryId) {
        ....
    }
....
}
```

EventBus

An EventBus is used to propagate application wide events in the GWT client layer. The EventBus is an instance of GWT's `HandlerManager` using the GWT event pattern introduced in GWT 1.6 (Overview).

There are two minor differences to standard GWT Event implementations:

- Each event defines its EventHandler interface as an inner interface named Handler

- The event type is a public static final field of the event class

Event example (details omitted)

```
public class UserNotificationEvent extends GwtEvent<UserNotificationEvent.Handler> {

    // Definition of the EventHandler as inner interface
    public interface Handler extends EventHandler {
        void onUserNotification(UserNotificationEvent event);
    }

    // Type of the Event when EventHandler is registered at a HandlerManager
    public final static Type<Handler> TYPE = new Type<Handler>();

    private final String notification;

    public UserNotificationEvent(String notification) {
        this.notification = notification;
    }

    public String getNotification() {
        return notification;
    }

    @Override
    public Type<Handler> getAssociatedType() {
        return TYPE;
    }

    @Override
    protected void dispatch(Handler handler) {
        handler.onUserNotification(this);
    }

}
```

Events in FrontOffice

The following table shows the events on the EventBus and their meaning within the application. Essentially, there are two types of events - events informing about event that happened (account was selected, adviser was updated) and events that require an action to be performed. *The later are currently under review and might be replaced by explicit calls to a Presenter*.

Note, that for example a contract note can trigger 2 events: an AlertEvent and a ContractNote event.

Event name	Description

AccountSelectedEvent	An account was selected, either as result of a search, selected the account in the account selection drop-down, clicking on a link or when restoring the state from the browser history. It holds the Id of the selected account plus the client which should be selected (Defaults to primary client after searches).
AlertEvent	Received a new alert from the server via Streaming.
AlertsExpiredEvent	Received for alerts that have been automatically removed after a certain period.
ClientSelectedEvent	Similar to `AccountSelectedEvent`, fired when a client was selected. Holds the Id for the Client's ClientRole.
ContractNoteEvent	Received a contract note for an account that currently logged in user has access to.
OrganisationNodeAddedEvent	Fired when a node is added to the Organisation structure administration.
OrganisationNodeDeletedEvent	Fired when a node is deleted in the Organisation structure administration.
OrganisationNodeUpdatedEvent	Fired when a node is updated in the Organisation structure administration.
UnseenAlertCountUpdatedEvent	When the number of unseen alerts has changed.
UpdatedUserEvent	Fired is a user was created or updated.
UserNotificationEvent	Show a notification to the user.
WorkspaceSelectedEvent	The currently shown workspace has changed. Carries the WorkspaceId of the newly shown Workspace.
PlaceChangedEvent	The state of one of the Presenters has changed and a new browser history token shall be issued. See Browser History for details.
SessionLostEvent	The authenticated session has expired (or server was restarted) and the client has received a HTTP status code 403.
PerformQuoteEvent	Switch to the 'Get Quote' workspace and show quotes for the specified security code(s).

SelectWorkspaceEvent	Show the specified workspace.
ShowMaintainUserEvent	Switch to the 'Users & Security' workspace and present the specified user.

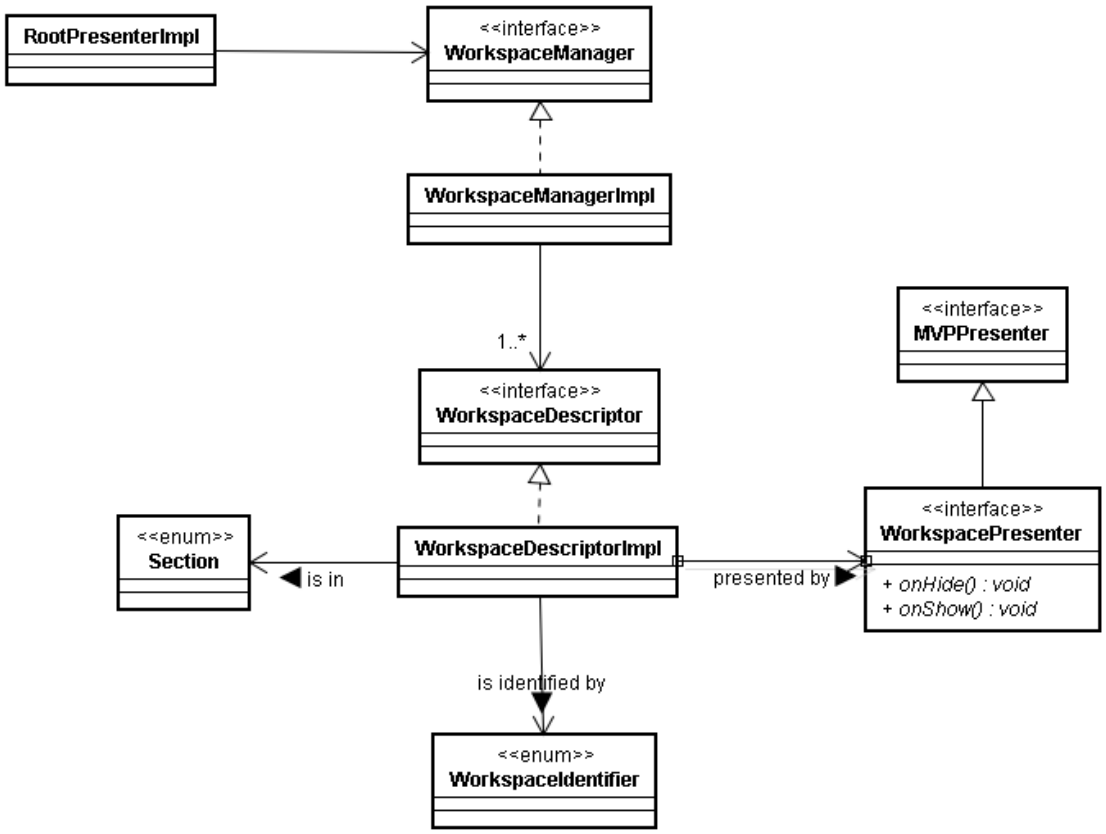
(core events are bold)

Sections, Workspaces and Applets

A section is an item in the top level navigation of the application, e.g. Adviser, Client or Admin. A section consists of multiple workspaces. A workspace is an item in the 2nd level navigation of the application. Workspaces can consist of 'Applets' or just be a single work area taking up all the available space on the screen.

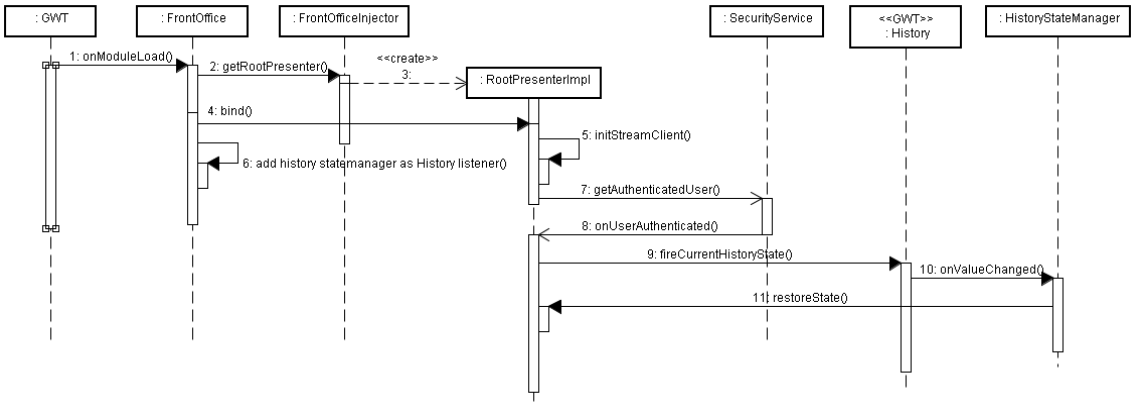
Workspaces and Applets are each implemented by a Presenter/Display pair. For both Section and Workspace enumerations exist that identify them (Note: this might change with a dynamic configuration of the structure).

As a basis for a later dynamic configuration of workspaces and sections, the definition is encapsulated by these classes:



FrontOffice client initialisation

The following diagram shows the outlined initialisation of the FrontOffice GWT module:

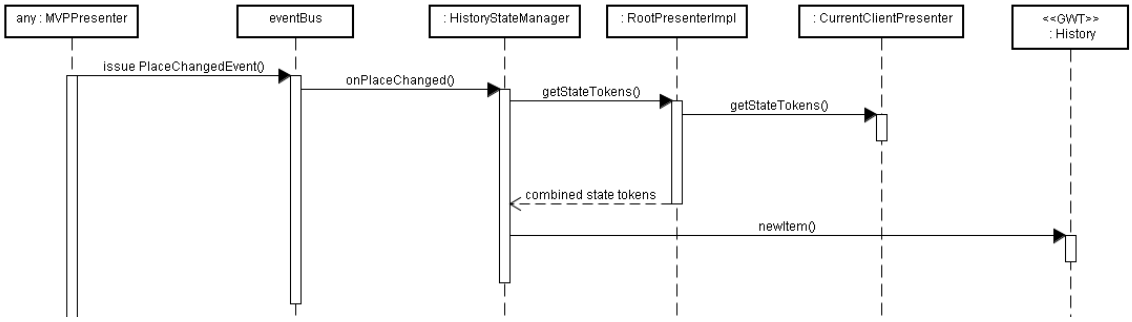


- `FrontOffice.onModuleLoad` is the application's entry point invoked by GWT
- (An uncaught ExceptionHandler is attached.
- An instance of `FrontOfficeInjector` for accessing the GIN managed instances is created and an instance of `RootPresenter` is created.
- `FrontOffice` added an instance of `HistoryStateManager` as GWT History changed listener.
- `RootPresenter.bind()` is invoked:
  - It initialises its child presenters as well as the Workspace manager's Presenters.
  - Initialises the streaming
  - Invokes `SecurityService.getAuthenticatedUser()` to retrieve details about the users authenticated on the server (by Spring Security)
  - Once the remote call returned,
    - Stores data about the authenticated user
    - Fires GWT's history resolution through `History.fireCurrentHistoryState()` which will be handled by `HistoryStateManager`. Refer to the section 'Browser History' for details about the implementation of this event.

Browser History

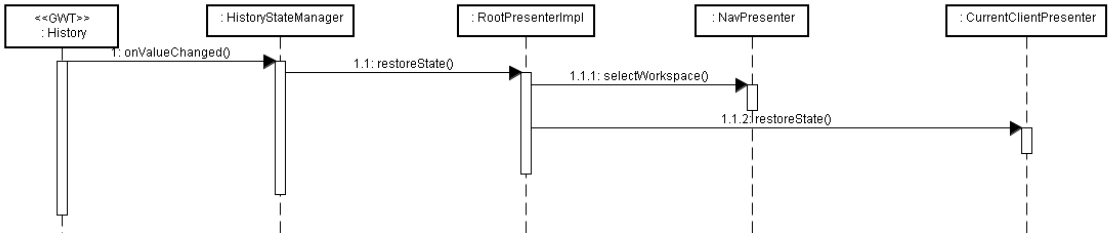
Browser History and therewith reload and forward/backward navigation by the browser as well as bookmarking is supported.

Every Presenter that changes the history state of the application (e.g. client selected) issues a `PlaceChangedEvent` onto the EventBus. A central event handler `HistoryStateManager` listens for these events, requests history state information from the application's `RootPresenter`. It is the RootPresenter's responsibility to get this history state information from each of its displayed components (e.g. Applets or Workspaces). A GWT History browser token containing this history state is issued then. Presenters that hold state for history implement the `HasHistoryState` interface.



Vice versa, the `HistoryStateManager` listens for changed history token events issued by GWT. It then parses

the token and forwards the parsed history tokens to the `RootPresenter` which then restores the application's state based on these tokens.



Note that the initial application state (after the User was authenticated) is restored by GWT's `History.fireCurrentHistoryState()` which issues a changed history token event on which `HistoryStateManager` is listening.

Forms and Validation

The application contains a lightweight support for validation, supporting the MVP concept. These classes are in the package `com.gbst.frontoffice.widgets.forms`.

The package contains the `CanDisplayError` interface that marks a UI component as being capable of displaying an error. This decouples the setting of an error message from its display, and allows the creation of multiple UI components that display errors in different ways.

The package contains an `ErrorBinder` interface which allows validation messages, created by Presenters, to be associated with arbitrary objects, which are usually input fields. This frees the presenter from knowing about the specifics of how error messages are displayed in the client.

`ErrorBinderImpl` is the standard implementation of `ErrorBinder` that maps an object to a `CanDisplayError`. Typically, a `Display` would instantiate the error bindings by calling `ErrorBinderImpl.bind` for all UI components that support validation. The `ErrorBinderImpl` is then exposed to the Presenter via the `ErrorBinder` interface.

In the case where there are multiple `ErrorBinders` in a display, a `CompositeErrorBinder` can be used to aggregate them. This allows the presentation of a single `ErrorBinder` to the presenter.

Example of usage for validating the field 'lastName' (code omitted and changed slightly for clarity):

Presenter

```
public class CreateClientPresenter extends
WorkspacePresenterImpl<CreateClientPresenter.Display> {

    /** Interface exposing HasValue<String> for the field and the
HasValidationFeedback interface */
    public interface Display extends MVPDisplay {
        HasValue<String> getLastNameField();
        ErrorBinder getErrorBinder();
        ...
    }

    /** Validate the form fields */
    private boolean isValid() {
        boolean valid = true;
        ErrorBinder errorBinder = getDisplay().getErrorBinder();
        errorBinder.clearAllErrors();
        if (!GwtStringUtils.hasText(model.getLastName())) {
            /** Dispay the error message */
            errorBinder.setError(getDisplay().getLastNameField(), "Please enter a
name");
            valid = false;
        }
  ...
        return valid;
    }

    private void onSave() {
        updateModelFromView(); // Transfer the current values in the model/DTO
        if (isValid()) {       // Validate and continue only if validation was
successful
     ....
        }
    }

}
```

Display

```
public class CreateClientDisplay extends Composite implements
CreateClientPresenter.Display {
    ...
    @UiField TextBox lastNameField;
    @UiField ErrorLabel lastNameError;  // implements CanDisplayError
    private ErrorBinderImpl errorBinder;

    public CreateClientDisplay() {
                ...
        errorBinder = new ErrorBinderImpl();
        errorBinder.bind(lastNameField, lastNameError);
  ...
    }

    public ErrorBinder getErrorBinder() {
        return errorBinder;
    }

    public HasValue<String> getLastNameField() {
        return lastNameField;
    }
}
```

Tabular Data Presentation

Tabular data is presented in many places in the FrontOffice application. An abstraction, `TableWidget` has been developed that makes common table operations such as column management, sorting and searching easier. `TableWidget` has two general purpose implementations, `FixedWidthGrid` and `TableWidgetImpl`. `FixedWidthGrid` provides a table where columns are of fixed width and cannot be adjusted by the user. In addition, the column headers are not sortable by user clicks on the header. `TableWidgetImpl` provides an implementation where column widths are user-adjustable and sorting can be user-initiated by clicking on the column header.

The `ColumnDefinition` interface to obtain metadata about column display. Typically, a particular usage of a `TableWidget` will have a corresponding class containing the columns to be displayed which extends `AbstractColumnDefinition`. `AbstractColumnDefinition` provides a pre-Java 5 style enumeration implementation which eases management of `ColumnDefinition` instances. Native Java 5 enums are not used as methods common to `ColumnDefinitions` were re-implemented for each enum. `TableWidget` columns are not referenced by index, but rather by a `ColumnDefinition` instance. On a call to `TableWidget.setColumns()` the `TableWidget` assigns column indexes to the supplied `ColumnDefnintions` in iteration order. Column indexes are then hidden from callers.

`TableWidget` allows sorting to be performed on multiple columns. When cell values are updated, an optional `Double` or `Date` can be specified along with the `String` display value which sets the value to be used for sorting. If such a value is specified, it will be used for sorting rather than the `String` display value. `TableWidgets` can be sorted programatically by calling the `sort(SortColumn<C>...)` method. Once sorted, the table does not automatically re-sort on data insertions or updates. It is up to the caller to re-sort data on table mutation. If the sort column have not been changed, a call to the `resort()` method will sort the table using the current sort columns.

`TableWidgets` support an optional identifier to be assigned to each row, using the `TableWidget.setIdOfRow(`

`)` method. Typically the unique key of the row would be assigned as an identifier. The row index of containing an identifier can be obtained using the `TableWidget.getRowWithId()` method. These allow callers to easily identifiy whether data is present in a table, and to identity the row to update if data has mutated and new values are to be displayed.

Common concepts

Application Security Model

FrontOffice uses Spring Security (formerly Acegi Security) for authentication and authorisation.

- Transport Layer Security
  All FrontOffice pages (login and the hosting HTML page) as well as remote procedure calls using SSL. If the user tries to access one of the URLs via HTTP, Spring Security redirects to SSL and also enforces a proper authentication.
  A servlet filter, `CustomHttpsRedirectFilter`, redirects any HTTP requests for HTTPS. Usually this would be implemented using Spring Security, but with being virtually unable to dynamically configure this through Spring Config (req'd for GWT's hosted mode testing), this configurable filter was implemented.
- Authentication
  The application's login page is external to the GWT application (faster initial load and high decoupling of authentication mechanism and application). Both the hosting HTML page and RPC calls require authentication and are intercepted by Spring Security.
  Not authenticated requests to the hosting page will be redirected to the configured login page, un-authenticated RPCs will return a HTTP status code 403. This custom behaviour is implemented in `RpcSe rviceAuthenticationEntryPoint`.
  In case of a timed out session, the application receives a status 403 from the remote service, a reload of the hosting page is triggered which then results in a forwarding to Spring Security's login mechanism. This decouples the GWT app from any authentication logic.
  User credentials are provided by a `UserDetailsService` implementation. FrontOffice user details are stored in the `SystemAccessRole`, storing User ID and a MD5 password hash.
- Authorisation
  Certain PartyRole of the domain model map to certain "Granted Authorities" allowing the usage of @Secured annotations on the ActionHandler implementations and other parts of the application.

PartyRole	Granted Authority	Comment
SystemAccessRole	ROLE_SYSTEM_ACCESS	Every user has this role as it is required to authenticate
AdminSectionAccessRole	ROLE_ADMIN_SECTION_ACCES S	Administrators, allowing access to the Admin section of the application

On the client side, an interface encapsulated the user's authorities, currently being implemented by the `RootPrese nter`.

Search Functionality

Overall - Search requests are made asynchronously via the command pattern implemented in FrontOffice. These searches may involve multiple sub searches that are (currently) executed synchronously. The results of these sub searches are collated and returned to the client.

Quick Search - The Quick Search executes an asynchronous SearchAction as the user enters criteria in the quick search textbox. When the search results return, they are displayed in the quick search results popup. The Quick Search executes the Client, Account and Quote sub searches via a SearchAction.

Advanced Search - The Advanced search executes an asynchronous Action when the user clicks on the search button (or the <Enter> key). When the search results return, they are displayed in the search results section. If the Advanced section **IS** populated, the search request (Action) is made via an AdvancedSearchAction and only executes an Account sub-search. If the Advanced section **IS NOT** populated (default scenario), the search request (Action) is made via a SearchAction and executes the Account, Account Designation, and Client sub-searches.

SearchAction sub-searches:

Account sub-search - The Account Search queries the FrontOffice database for accounts that match the search criteria (based on account name, or account number, and are viewable by the user.

Client sub-search - The Client Search queries the FrontOffice database for clients that match the search criteria (based on party name), and are viewable by the user.

Quote sub-search - The Quote Search queries a Market Information Provider (ASX) for price quotes of security codes. These are then returned in the results.

Account Designation sub-search - The Account Designation Search queries the Shares database for accounts that match the account designation, but the results only include those accounts also in the FrontOffice database.

AdvancedSearchAction sub-searches:

Account sub-search - The Account Search queries the Shares and DCA databases for accounts that match the search criteria (based on account name, account number, or securities and (optionally) the number of units held), and are viewable by the user.

Alerts

Alerts are text messages directed at a user or group of users informing them of some event. An alert can be optionally be linked to an account, making this information available to the displayer of alerts can enable navigation to the account at the source of the alert.

Each alert can have a number of alert receipts. Each alert receipt indicates a user to whom the alert is targeted. Alert receipts also indicate whether the targeted user has viewed the alert.

When an alert, and its associated receipts are created, a Stream Message is also created for each of the recipients and sent to each of their connected sessions, allowing clients to be updated immediately an alert is created. The stream messages sent to the client can contain an optional payload that contains information related to, but not part of, the alert. In the case of contract notes, both the alert, containing the summary text, and the contract note are sent is a single `StreamMessage` allowing the client to add the alert to the alert tables, the contract note to the contract note display (if the user is viewing the same account), and display a pop-up containing the contract note details.

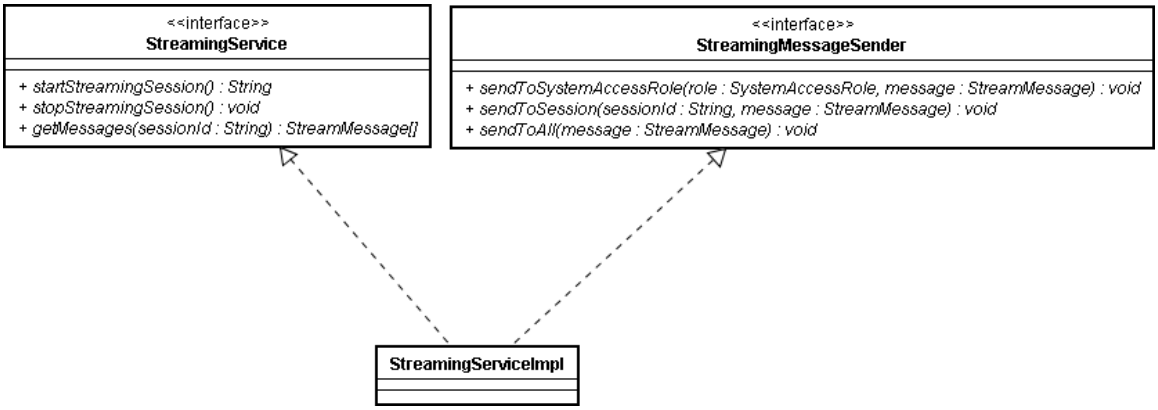
Alerts are retained for 14 days, after which time they are expired by deleting the alerts and their associated alert receipts. A streaming message is sent to connected clients informing them of the alerts that have been expired, allowing the client to remove those alerts from the display.

Streaming

The FrontOffice server is able to push, or stream, data to the client without the client explicitly requesting this data. This allows clients to display updates to users with low latency, in near real-time.

Streaming is accomplished by using a long-poll technique. Clients use a HTTP request to poll the server for messages. The server either sends any outstanding messages immediately to the client, or in the case where there are no messages, keeps the connection open for a long period. If a message for the client arrives in this period, it is sent to the client and the connection closed, otherwise, the connection is closed after the specified duration with no messages. Once the connection has been closed, the client polls the server again for further messages.

The following class diagram shows the server side components of the streaming service.



The `StreamingServiceImpl` implements two interfaces, `StreamingService` to provide a client interface to the streaming service and `StreamingMessageSender` for other server side components to send messages to clients.

The `StreamingService` is exported to clients by the standard GWT RPC mechanism. When clients call `startStreamingSession` the server generates a unique identifier for the session which is returned to the client. The server creates a queue for each connected session which stores messages to be sent to the session. When clients call the `getMessages` method, the messages in the queue are serialized and sent to the client.

The `StreamingMessageSender` exposes methods to server side components that need to send messages to clients. Messages can be sent to all connected users, all sessions for a single user or a single session.

The streaming service supports an extensible messages. All messages must implement the `StreamMessage` interface and must also be serializable. Client components wishing to handle stream messages must register a `StreamClientMessageHandler` with the `StreamClient` for a specific class of `StreamMessage`. The `StreamClient` will call back the registered handlers when messages are received.

Server concepts

Integration of Jasper Reports

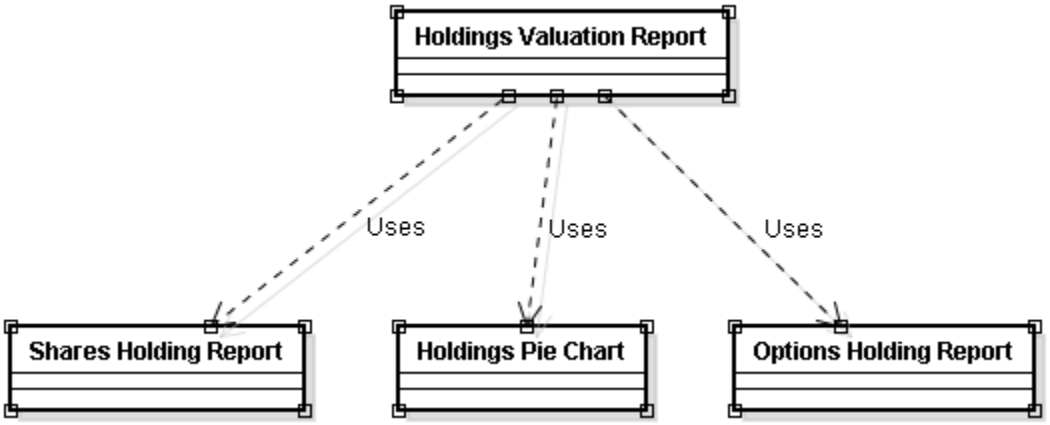
Jasper Reports is a leading Java reporting engine. FrontOffice uses the Jasper Reports reporting engine to generate reports to pdf files.

When a user requests a report, the report is populated with a datasource generated using existing FrontOffice functions, and then generated. A unique reference to the file is returned to the client, which then calls a Servlet with this reference and the pdf file is returned. The Servlet exists to present reports (pdf files) to logged in users who are permitted to view them.

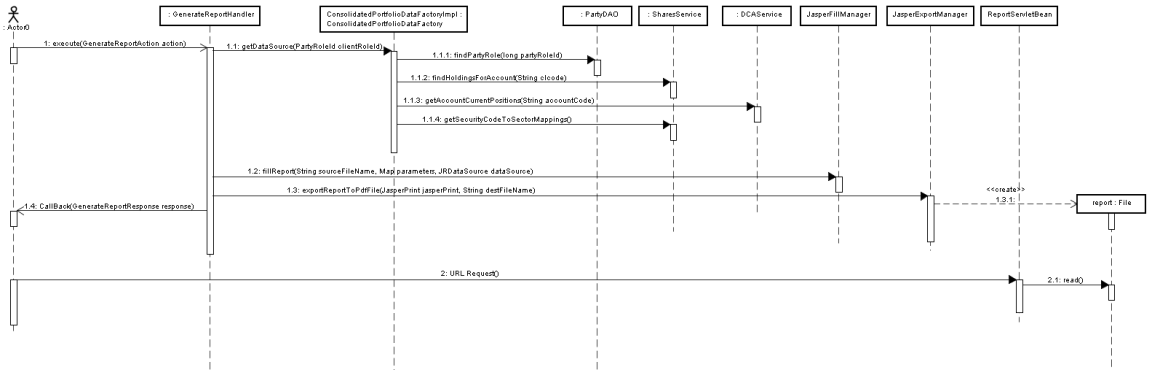
JasperReports uses a JRDataSource to populate its reports. FrontOffice provides implementations of JRDataSource as the DataSource, containing either additional JRDataSources or core Java classes. This is to enable loose coupling between JasperReports and FrontOffice.

Similar to other reporting frameworks (eg Oracle Reports, Crystal Reports), Jasper Reports supports Subreports. Subreports represent one of the most advanced feature sets of JasperReports, and they enable the design of very complex reports. They enable reuse of components, and even complete reports. With complete reports able to be used as subreports, it makes sense to only store report components (eg address blocks) in a subreports directory, whilst other reports stay at a primary level. Images used in the reports are stored in a "images" subdirectory and referenced at runtime.

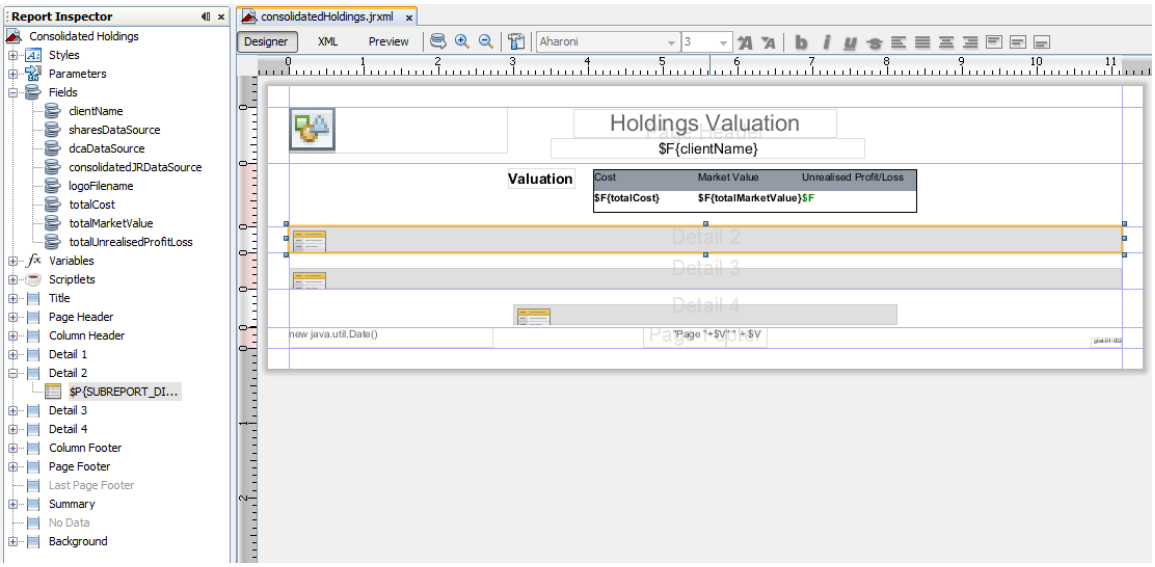
An example of this is the construction of the Holdings Valuation Report which utilises three subreports.



Report Sequence Diagram



To enable rapid development/maintenance of reports, the WYSIWYG tool iReport (provided and supported by the creators/maintainers of JasperReports, JasperSoft) is used.



Databus

The FrontOffice application is notified in real-time about events from the Shares back office via the Shares Databus. The Databus is a JMS topic on which changes in the Shares database are published. The messages on this topic are XML representations of table rows in the Shares database. The FrontOffice application subscribes to this topic and can take action based on the messages received.

The application subscribes to the Databus topic using a durable subscription, which means that messages delivered to the queue while the FrontOffice application is not connected to the topic will be saved and delivered once the application re-connects. Durable subscriptions require an identifier that is unique across subscribers, but persistent across reconnections. The FrontOffice application uses a hash of an ordered list of the network addresses of all network interfaces on the running hardware to obtain a subscription id. This subscription id will be stable for a given hardware configuration unless network interfaces are added or removed.

The FrontOffice application uses a JMS message selector to filter messages that the application does not process. Currently, only contract note creation messages are processed. Upon receipt of a contract note creation message, all users who have access to the account are determined. If there are any users, an Alert is created and persisted. This alert, together with a serialized representation of the contract note is sent to all connected clients who have access to the account via a streaming message. The connected clients can then add the alerts to the alerts table and display a pop-up with the contract note details.

Scheduled Jobs

FrontOffice uses Quartz for scheduling job through Spring's Quartz integration.

A Quartz factory bean as well as the scheduled jobs and their triggers are defined in `beans-jobs.xml` which is part of the Spring application context.

Currently these jobs are defined

Job name	Scheduled time	Description

ExpireAlertsJob	daily at 12:30am	Removes all alerts older than 14 day and propagate the removal of the alerts to the clients via Streaming Implemented in `ExpireAlertsJob`
PublishIndexMarketInfoJob	daily every 30 seconds	Retrieves the pricing information from a `MarketInformationProvider` implementation and published the updated pricing to all clients via Streaming. Implemented in `IndicesMarketInformationPublisher`

Commissions Calculator

The Commissions Calculations are provided by a Commissions Helper class. The helper class also provides information on number of trades. There are 3 public methods for this class:

```
List<ClientRevenueDTO> getCommissionsForClient()
```

Utilises the summarised table PUB."activity" from the Shares database, and the stored procedure fo_sp_GetGroupEODTopClients from the DCA database to obtain the clients, and the monthToDate, previousMonthToDate and yearToDate commission and "number of trades". This is restricted to the accounts that an adviser/user has permission to view. The commission and trade data is then merged at the client level.

```
Map<String, double[]> getCommissionsForAdviser()
```

Utilises the summarised table PUB."activity" from the Shares database, and the stored procedure fo_sp_GetGroupEODBrokerageStatistics from the DCA database to obtain the monthToDate, previousMonth and yearToDate adviser commission that an adviser has received (based on the adviser codes that an adviser/user has permission to view). The stored procedures fo_sp_GetGroupEODTopClients and fo_sp_GetGroupEODBrokerageStatistics use different queries to determine the commission values, and therefor may differ in their results.

```
List<TemporalRevenueAndContracts> getMonthsCommissionsForAdviser(int numberOfMonths)
```

Utilises the summarised table PUB."activity" from the Shares database to obtain the adviser commission and "number of trades" for each of the last "numberOfMonths" (inclusive of current month). Results are restricted to accounts that an adviser has permission to view. This only reports on the shares data, no such functionality is available on the dca database.

Module 'shares-facade'

The structure and design of this module is identical to the module `dca-facade`.

This module exposes a single interface `SharesService` containing the required methods for the FrontOffice. The methods of this class return Shares specific classes (POJOs) and accept Shares specific parameters like clCode. All of the queries are written as iBatis SQL queries and no stored procedures are used.

The implementation of SharesService, SharesServiceImpl, uses iBatis as OR mapping tool and extends Spring's SqlMapClientDaoSupport which wraps iBatis support. iBatis is configured through the the file shares-sql-map-config.xml which references SharesService.xml that defines the queries.
Transaction boundaries are defined in the SharesService interface through Spring's @Transactional annotation.
The definition of the corresponding Spring beans SharesService, SharesServiceImpl and their datasource is in beans-shares.xml in the client module. Currently the datasource is defined within the Spring configuration as a **not pooled** datasource (not as a datasource within the application server).

Package Structure

com.gbst.frontoffice.shares	Contains only the interface SharesService
- data	Shares specific POJOs (part of the module's public interface), parameters and result objects of SharesService.
- impl	Implementation of SharesService and any implementation specific classes.
- typehandler	iBatis typehandlers for converting database values to Java enumerations.

Module dca-facade

The structure and design of this module is identical to the module shares-facade.

This module exposes a single interface DCAService containing the required methods for the FrontOffice. The methods of this class return DCA specific classes (POJOs) and accept DCA specific parameters like accountCode or groupCode. Most of the queries use stored procedure provided by the DCA database (prefix with fo_sp), but some of the searches are implemented as SQL queries in iBatis.

The implementation of DCAService, DCAServiceImpl, uses iBatis as OR mapping tool and extends Spring's SqlMapClientDaoSupport which wraps iBatis support. iBatis is configured through the the file dca-sql-map-config.xml which references SharesService.xml that defines the queries and stored procedure calls.

Transaction boundaries are defined in the DCAService interface through Spring's @Transactional annotation.
The definition of the corresponding Spring beans DCAService, DCAServiceImpl and their datasource is in beans-dca.xml in the client module. The datasource is defined within the Spring configuration (not as application server datasource).

Package Structure

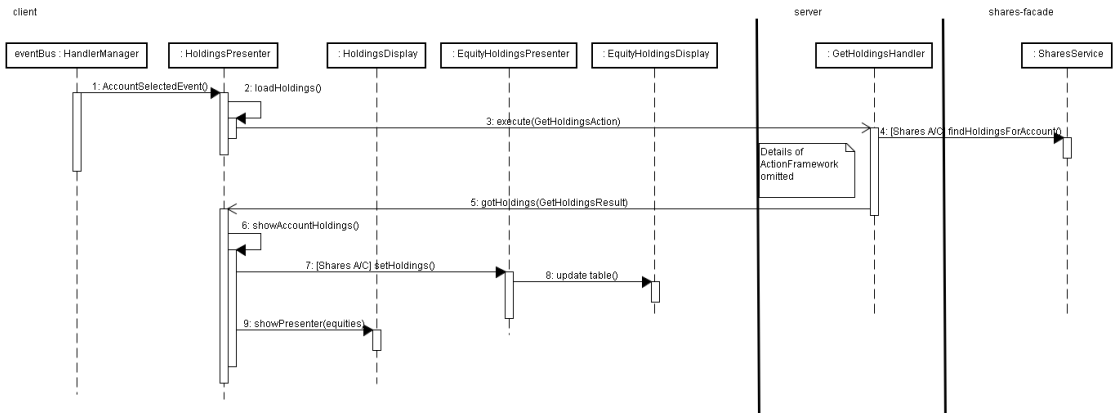
com.gbst.frontoffice.dca	Contains only the interface DCAService
- data	DCA specific POJOs (part of the module's public interface), parameters and result objects of DCAService.

- `impl`	Implementation of `DCAService` and any implementation specific classes.
- `typehandler`	iBatis typehandlers for converting database values to Java enumerations.

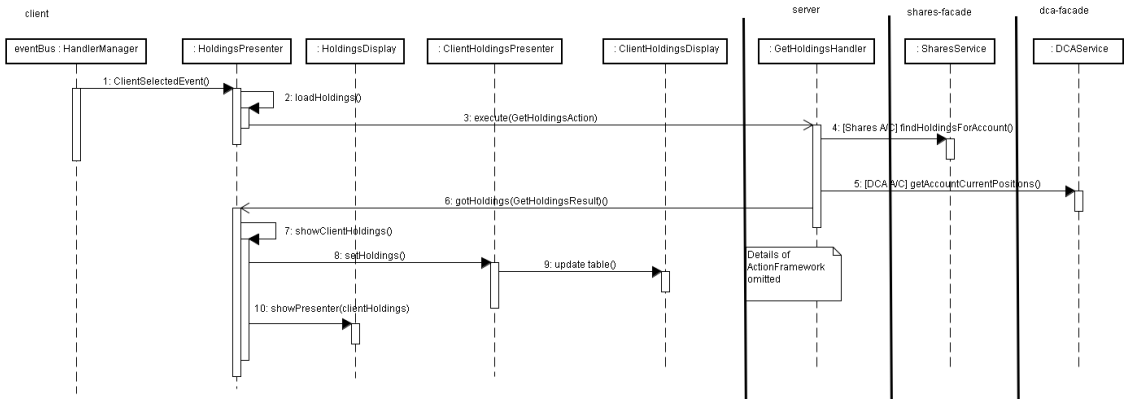
Use case realisation : Display Holdings

This section gives an overview of the implementation of the 'Display Holdings' use case. It shows two simplified sequence diagrams across all modules. One diagram shows the displaying holdings of a Shares account (DCA would be identical except for the backening service), and one displaying consolidated Holdings gathering data from multiple accounts.

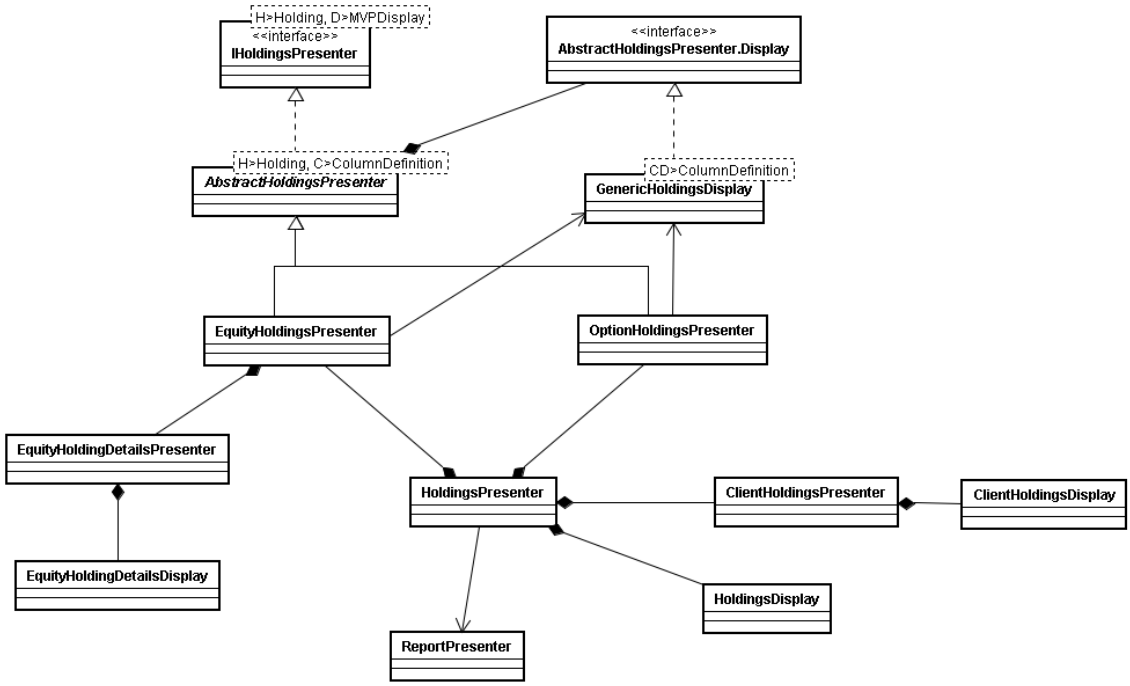
Display Equity Holdings



Display Consolidated Holdings



Structure of HoldingsPresenter



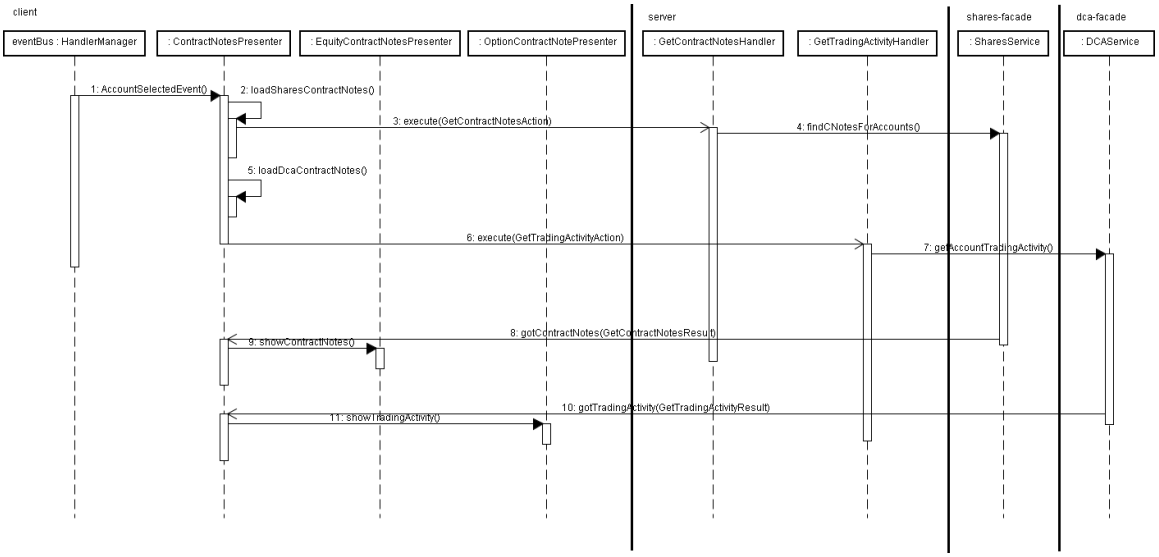
The structure above shows abstracts for Equities and Options holdings. Both EquityHoldingsPresenter and OptionHoldingsPresenter extends a common AbstractHoldingsPresenter implementation which implements common strategies. Both presenters use the same generic Display implements `GenericHoldingsDisplay`.
ClientHoldingsPresenter and *Display are implemented separated as it caters for a summary table and tables for equities and options with limited columns.
The structure shows that rule that presenters only know presenters as well as the way of factoring out common behaviour in abstract superclasses.

Contract Notes

The contract notes workspace combines data sourced one or both back offices. Contract notes are sourced by querying the shares database and options trading activity is obtained through a stored procedure in the DCA database.

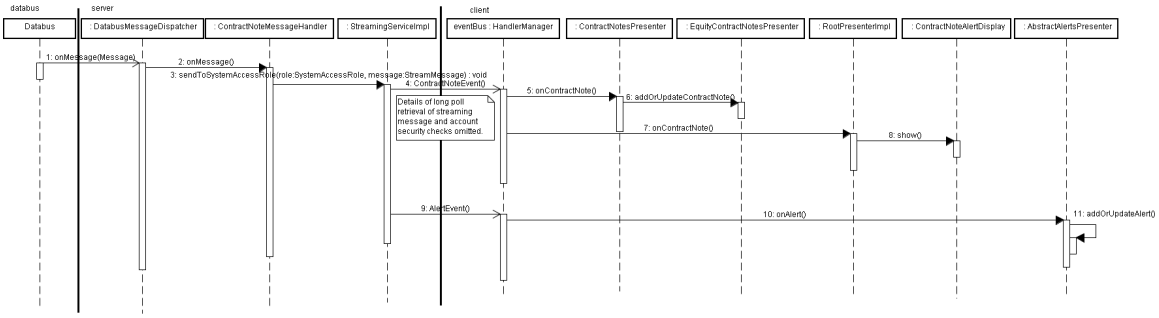
Display contract notes

The following sequence diagram shows the mechanisms used to display contract notes and trading activity.



Streaming Contract Notes

The following sequence diagram show the how a contract note created in Shares is displayed in real-time in the Front Office application.



The processing sequence is:

1. Messages are retrieved from the Databus topic by the `DatabusMessageDispatcher`.
2. The `DatabusMessageDispatcher` then notifies its listeners of a new message arrival. Listeners may choose to ignore or process the message, depending on the message properties. In this case, the `ContractNoteMessageHandler` processes the contract note message.
3. The `ContractNoteMessageHandler` determines whether there are any users that should receive the message, based on account security policies. If so, an alert is created and persisted and the recipients notified of the alert and a serialized representation of the contract note via the `StreamingServiceImpl`.
4. The client uses a long poll mechanism to retrieve messages, and receives a `ContractNoteAlertMessage`. A `StreamClientMessageHandler` creates a `ContractNoteEvent` and an that is dispatched to event handlers on the client.
5. The `ContractNotesPresenter` is notified of the contract note.
6. The `EquityContractNotesPresenter` is notified of the contract note. If the account on the contract note is currently displayed, then the table of contract notes is updated with the new contract note.
7. The `RootPresenterImpl` is notified of the contract note.
8. The `RootPresenterImpl` shows the details of the contract note via the `ContractNoteAlertDisplay`.
9. A `StreamClientMessageHandler` creates an `AlertEvent` that is dispatched to event handlers on the client.
10. Instances of `AbstractAlertsPresenter` are notified on the `AlertEvent`.
11. Instances of `AbstractAlertsPresenter` update their alerts table and may perform other actions, such as

updating the unseen alert count.

Build process / Maven

The application is built using Maven 2.

The Maven module structure represents the modules as described in 'Module Structure'. The project is defined via Maven's 'Project Aggregation', with a parent POM defining all plugins and dependencies as well as the modules belonging to this project. It also defines various reports that can be generated via `mvn site`.

Reports:

- Cross reference source, used by other report plugins `maven-jxr-plugin`,
- Unit test report `maven-surefire-report-plugin`
- Unit test coverage via Cobertura `cobertura-maven-plugin`
- PMD source code analyzer `maven-pmd-plugin` (rules not fully customised yet)
- Findbugs report `findbugs-maven-plugin` (rules not fully customised yet)
- Taglist plugin extracting TODO and similar tags `taglist-maven-plugin`

Except for the client module (see below), the POMs of the other modules are standard Java POMs without any additional build plugins.

Client module

The client module uses plugin for GWT and Jasper Reports compilation.

Jasper Reports

Reports (*.jrxml) are compile via the `jasperreports-maven-plugin` in the phase `generate-resources`. The resulting *.jasper files and images are copied into the `target/classes/reports` directory via the standard maven resource plugin.

GWT

The used plugin for GWT integration is `gwt-maven-plugin`. For hosted mode the resulting GWT classes and resource are copied into the `src/main/webapp` directory. This allows in-place editing for CSS as well as seamless refreshing of changed classes.

Within the client module, GWT's hosted mode can be started via

```
mvn gwt:run
```

or in debug mode (remote debugger to attach on port 7000):

```
mvn gwt:debug
```

Releases

FrontOffice releases are built directly through Maven. Refer to Releasing and Deploying

Local repository

The project uses a Livewirelabs Maven repository that contains depedencies which are not available in other public repositories (e.g. GIN, MS SQL Server JDBC driver, etc).

The LWL repository is hosted a in subversion and has the URL http://code.livewirelabs.net.au/svn/mavenrepo. Please refer to DevEnvConfig to setup Maven for accessing this repository.

Releasing FrontOffice

Building a deployable WAR outside the scope of a release

Run:

```
mvn -Prelease clean install
```

This will create 2 artifacts (see below), the deployable WAR file and an archive containing ANT scripts to create the database.

Building a release with Maven

FrontOffice can be released by using the standard Maven release process (mvn release plugin guide), although copying the released WAR file to the Application server is not configured.
This process:

- sets the release version
- compiles and runs all the tests
- adds the appropriate svn tag
- progresses the svn trunk to the next snapshot version

Commands to perform:

```
mvn release:prepare
mvn release:perform
```

Important note

Due to a svn/mvn bug, mvn release:prepare requires a svn command line version >=1.6.5

Artifacts created by the build process.

All artifacts (WAR and the databases zip files) are deployed to the Livewire Nexus repository, starting with v0.2.2.

Web application archive (WAR)

A standard WAR is created by the build process in `client\target` and is named `frontoffice-<version>.war`. It has a context path set to /.

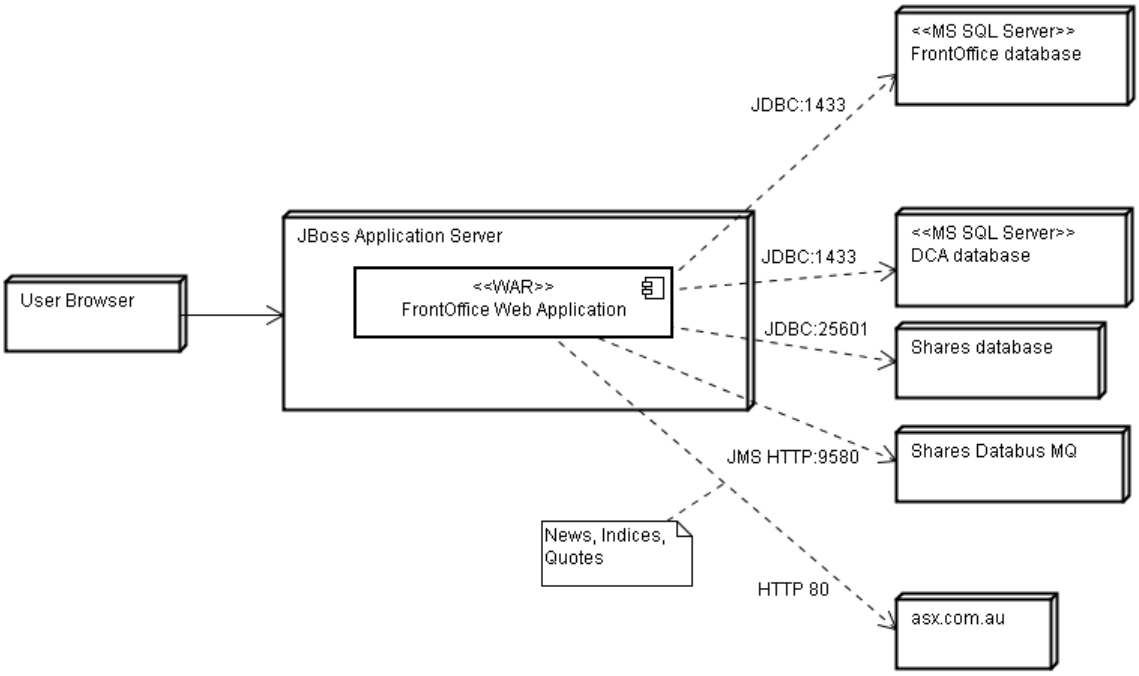
Database release

The database release consists of 3 archives, one for SQLServer installations (batch scripts), two for Postgresql installations (batch and shell script versions). The archives can be found in `database/target/` and are named `frontofficedb-<version>~~createdb-postgresql.(zip|tar.gz)~~` and `frontofficedb<version>-createdb-sqlserver.zip`.

Deployment Model

Overview

The following diagram shows the FrontOffice deployment structure and dependencies:



The deployed WAR file is the resulting artifact of the build of the client module.

JBoss 5 configuration

Used version: JBoss 5.1.0.GA JDK1.6

Manual installation steps after JBoss installation:

1. Copy `eclipselink.jar` and `sqljdbc.jar` into `server/default/lib`
2. JBoss wants to use its own JPA providers and will not leave JPA instantiation to Spring. To fix, edit `server/default/deployers/ejb3.deployer/META-INF/jpa-deployers-jboss-beans.xml` and comment out the following three beans:
   `PersistenceParsingDeployer`
   `PersistenceDeployer`
   `PersistenceUnitDeployer`
3. SSL config
   a. Uncomment/change this line in `server/default/deploy/jbossweb.sar/server.xml`

```
<Connector protocol="HTTP/1.1" SSLEnabled="true"
     port="8443" address="${jboss.bind.address}"
     scheme="https" secure="true" clientAuth="false"
     keystoreFile="${jboss.server.home.dir}/conf/server.keystore"
     keystorePass="foappserver" sslProtocol = "TLS" />
```

   b. Generate a test certificate in a new keystore `server.keystore` and save it in `\server\default\conf`
   Keystore password must match keystorePass in the above XML and the key password should match the keystore password.

"Commands for generate a certificate"

```
keytool -genkey -alias tc-ssl -keyalg RSA -keystore server.keystore
```