Оглавление

[Архитектура серверной части приложения 1](#_Toc423460220)

[3 слоя приложения 1](#_Toc423460221)

[Уровень доступа к данным 1](#_Toc423460222)

[Уровень сервисов 2](#_Toc423460223)

[Уровень web-публикации 2](#_Toc423460224)

[Последовательность отработки web-запроса 2](#_Toc423460225)

[Требования к реализации 2](#_Toc423460226)

[Требования потоко-безопасности 2](#_Toc423460227)

[Требования к реализации компонентов 3](#_Toc423460228)

[Типы Виджетов 4](#_Toc423460229)

[Реализация инфраструктуры приложения 4](#_Toc423460230)

[Кастомизация используемых фреймворков 4](#_Toc423460231)

[Utilities 5](#_Toc423460232)

[Используемы техники 6](#_Toc423460233)

# Архитектура серверной части приложения

Приложение реализовано на базе фреймворка Spring-MVC с использованием компонентов платформы Spring-Context и Spring-Security. На уровне слоя данных используется фреймворк MyBatis и его модуль интеграции с фреймворком Spring.

## 3 слоя приложения

* уровень доступа к данным (@Repository, @Component)
* уровень сервисов (@Service)
* уровень web-публикации сервисов (@Controller)

Связывание между уровнями происходит через механизм Dependency Injection реализованный контейнером Spring-Context.

### Уровень доступа к данным

представлен в свою очередь двумя прослойками:

* уровень Мапперов – интерфейсов доступа к sql-данным, которые MyBatis связывает с sql-запросами (@Component, \*Mapper.java)
* уровень Репозиториев, которые предоставляют Сервисам удобный интерфейс доступа к данным (@Repository, \*Repository.java)

Мапперы обеспечивают прозрачное (1:1) отображение мира sql на мир java-объектов. Репозитории в большинстве случаев просто транслируют методы доступа своим клиентам, но также и предоставляют дополнительные методы «удобства» доступа к реляционным данным.

### Уровень сервисов

включает в себя две категории сервисов:

* Сервисы Фильтров предоставляют полный набор методов для создания Фильтров каждой отдельной Витрины (@Service, \*Filter.java)
* Сервисы Виджетов предоставляют набор методов для создания отдельного Виджета Витрины (@Service, \*.java)

На уровне Сервисов реализуется бизнес-логика преобразования реляционных данных, полученных из Репозиториев в требуемое для каждого Фильтра/Виджета представление.

### Уровень web-публикации

представлен:

* Универсальный FrontApiController (@RestController), принимающий все web-запросы к бизнес-данным (к Фильтрам и Виджетам) и направляющий их соответствующим Сервисам
* Узкоспециализированный ReportController (@Controller), задача которого выгружать по запросу клиента Excel-файл с подготовленным Сервисом Виджета отчета

FrontApiController не реализует никакой бизнес-логики и является тонким посредником между Сервисами и web-запросами к ним. Фактически он обеспечивает web-публикацию Сервисов с использованием механизма отображения входящих/исходящих данных из/в JSON.

## Последовательность отработки web-запроса

* FrontApiController получает запрос, из URL которого он определяет требуемый Сервис и его требуемый метод, затем получает экземпляр этого Сервиса от Spring-контейнера. Также из запроса Контроллер извлекает параметры вызова данного сервиса/метода и выполняет его вызов.
* Сервис, при инстанциировании Контекстом получает ссылку на требуемый Репозиторий или Репозитории. Сервис отрабатывает вызовы FrontApiController-а, обращаясь за данными к Репозиториям, преобразовывая полученные данные в требуемое бизнес-представление.
* Репозиторий, при инстанциировании Контекстом получает ссылку на требуемый Маппер, к которому он обращается, отрабатывая вызовы Сервиса.
* Мапперы создаются инфраструктурой MyBatis и регистрируются в Контексте в момент старта приложения. MyBatis создает их как прокси-объекты, которые отрабатывают поступающие к ним вызовы через выполнение привязанных к ним sql-запросов с последующей материализацией полученных данных.
* FrontApiController получив успешный ответ по вышеописанному стеку вызовов возвращает его клиенту как результат web-запроса. Инфраструктура MVC при этом преобразует его в JSON представление.

## Требования к реализации

### Требования потоко-безопасности

Все описанные компоненты приложения (Контроллеры, Сервисы, Репозитории) являются фактическими singleton-ами, будучи зарегистрированными в Контексте по умолчанию таковыми. Такой подход позволяет использовать единственный экземпляр каждого класса для обслуживания всех запросов. Это помогает повысить масштабируемость приложения.

Для обеспечения возможности одновременного использования одного экземпляра класса, его код должен быть реализован потоко-безопасно. Простейший способ достижения этой цели – сделать класс immutable (неизменяемым), т.е. устанавливать состояние экземпляра в момент его создания без возможности его изменения внутри его жизненного цикла. Средство решения – делать все поля final, а также не менять состояния внутренних reference-объектов (массивов, списков).

Если все же необходимо иметь некое общее состояние, разделяемое различными запросами, то изменение такого состояние необходимо синхронизировать через блокировку доступа. Примером разделяемого состояния может быть внутренний кэш некоторых «затратно-создаваемых» объектов.

При необходимости реализации нетривиальной последовательной бизнес-обработки данных в Сервисе может возникать желание складывать промежуточные результаты в поля класса. Такой подход разрушит потоко-безопасность данного класса. Во избежание такого исхода приходится передавать промежуточное состояние в параметрах при последовательном вызове внутренних методов. При большом количестве параметров таких методов можно создавать внутренний класс-хранитель таких промежуточных данных и передавать его экземпляр между вызовами. Также может упростить эту проблему вынесение части кода в отдельный класс-хелпер, особенно в случае наличия дублирующей логики необходимой в реализации нескольких Виджетов одной Витрины или смежных Витрин одного Субдомена.

При реализации классов хелперов и утилит, которые используются потоко-безопасными классами также необходимо придерживаться принципов потоко-безопасности, избегая изменяемого состояния (mutability).

### Требования к реализации компонентов

#### Мапперы

Мапперы, являясь интерфейсами, не имеют java-реализации, но имеют связанные с собой xml-файлы с текстом sql-запросов. Рекомендуется придерживаться подобной практики отделения кода sql от java-кода, помещая его в отдельные xml-файлы.

Мапперы не помечаются аннотацией, т.к. они автоматически «находятся» инфраструктурой *MyBatis* в момент конфигурирования на старте приложения. Их адресация происходит через указание в конфигурации Контекста для бина *MapperScannerConfigurer* пакета, в котором находятся данные интерфейсы. В свою очередь, xml-представления Мапперов должны храниться в ресурсах приложения по пути совпадающим с пакетом «своего» интерфейса.

Благодаря использованию модуля интеграции инфраструктуры *MyBatis* с инфраструктурой *Spring*, использование Мапперов становится очень простым. Нет необходимости заботится об sql-сессиях и подключениях – *MyBatis* с использованием технологии AOP автоматически создает и открывает соединение с БД и транзакцию в том методе, где используется Маппер, а также заботится об освобождении использованных ресурсов. Эта возможность включается добавлением бинов *SqlSessionTemplate* и *DataSourceTransactionManager* в конфигурацию Контекста. Для сценариев одностороннего использования БД (только чтение данных), которое преобладает в данном BI-приложении такого «пассивного» подхода достаточно.

#### Репозитории

Не имеют базового класса для наследования. Содержат набор методов «удобных» для использования Сервисами. «Удобства» включают в себя дополнительное сужение критериев запроса к данным, создание инвариантов вызова одного и того же метода подлежащего Маппера или пере-использование других методов данного Репозитория.

Как правило, для каждой Витрины создается один Репозиторий для получения данных Сервисов Фильтров и один для получения данных всех Сервисов Виджетов. Один Репозиторий Фильтра может быть совместно используемым несколькими Витринами, входящими в один Субдомен.

Классы Репозиториев помечены аннотацией @Repository, что вызывает их регистрацию в Контексте в момент старта приложения.

#### Сервисы Фильтров

Не имеют базового класса для наследования. Содержат набор методов, возвращающих массивы значений Измерений (в терминологии DWH). Некоторые из таких Измерений могут иметь зависимости друг на друга, например, быть иерархическими (Регион-Город-Отделение). В этом случае методов может принимать параметры ограничивающие «родительские» узлы. Принимая во внимание два факта:

* методы Сервисов Фильтров вызываются FrontApiController-ом через рефлексию
* Java 1.7 не сохраняет в byte-коде имен параметров

приходится дополнительно «навешивать» имена параметров через аннотацию *@Param*.

При вызове со стороны клиента URL составляется как *api/{filter}/{method}*, где filter – имя класса Сервиса и method – имя требуемого его метода без префикса «get», например, *api/cardsDeliveryPeriodFilter/regions*. Именованные параметры вызова метода помещаются в query-string web-запроса стандартным образом.

Классы Сервисов Фильтров помечены аннотацией @Service, что вызывает их регистрацию в Контексте в момент старта приложения.

#### Сервисы Виджетов

Наследуют базовому классу *BaseWidget*, в котором определен абстрактный единственный метод *getData*, принимающий объект с параметрами Фильтра Витрины (\*Options) и возвращающий требуемое бизнес-представление данных для создания одного Виджета Витрины. Этот метод вызывается FrontApiController-ом статически без рефлексии.

При вызове со стороны клиента URL составляется как *api/{widget},* где widget – имя класса Сервиса, например, *api/cardsDeliveryPeriodPie*. Значения параметров вызова метода *getData* помещаются в body web-запроса как JSON-представление соответствующего класса \*Options, реализованного для данной Витрины.

Каждая Витрина имеет собственный класс параметров Фильтра \*Options, который наследует базовому *BaseOptions*. Поля этого класса заполняются выбранными пользователем значениями в Фильтре Витрины. В терминологии DWH набор этих значений определяет требуемый Срез по Измерениям бизнес-данных. Все Виджеты одной Витрины использую один класс параметров данной Витрины.

Классы Сервисов Виджетов помечены аннотацией @Service, что вызывает их регистрацию в Контексте в момент старта приложения.

### Типы Виджетов

На Витринах используются следующие типы виджетов:

* Виджеты-диаграммы (наследуют BaseChart)
* Виджеты-таблицы (наследуют BaseWidget)
* Виджеты-отчеты (наследуют BaseReport) – выгрузка данных в Excel

## Реализация инфраструктуры приложения

### Кастомизация используемых фреймворков

#### MyBatis

Для обеспечения бесшовного использования приложением классов JodaTime в коллекцию TypeHandler-ов MyBatis-а добавлены 2 хандлера для маппинга новых типов данных.

Также MyBatis «научен» сериализовать массивы простых числовых типов и строк в строковое значение sql-параметра с разделителями запятыми. Этот подход поддерживается на стороне БД при вызове бизнес-функций pl/sql пакетов с параметрами, ожидающими такие списочные значения.

Все дополнительные хандлеры зарегистрированы в xml-конфигурации MyBatis с обязательным указанием их jdbcType (иначе этот тип приходится явно указывать в теле sql-запросов).

#### MVC Rest Json (Jackson)

Для обеспечения бесшовного использования приложением классов JodaTime в коллекцию де-/сериалайзеров добавлены 4 класса для отображения новых типов данных.

Подобным образом Jackson может быть «научен» сериализовать кастомные типы данных удобным способом, как, например, это реализовано для класса Color.

Необходимые настройки MVC-фреймворка выполняются через введение в конфигурацию Контекста бина *MappingJackson2HttpMessageConverter*, а также фабрики *Jackson2ObjectMapperFactoryBean*.

### Utilities

#### Linq – работа с последовательностями

Для упрощения работы с последовательностями объектов (сортировка, отбор, группировка и прочее) в рамках кода приложения реализована библиотека, называемая по аналогии с известной в .Net Framework как Linq.

Библиотека предоставляет набор статических методов для работы с последовательносями, каждый из которых принимает в качестве параметра аналог lambda-функции – функциональный объект, являющийся анонимной реализацией одного из стандартных интерфейсов, входящих в библиотеку.

Более удобное использование функционала библиотеки производится посредством изначального заворачивания исходной последовательности в специальный *LinqWrapper*, который затем позволяет вызывать стандартные методы библиотеки непосредственно как экземплярые. Такой подход позволяет реализовать функциональный стиль программирования с т.н. fluent-интерфейсом.

Стандартные интерфейсы: Predicate, Selector, BinaryOperation, Action.

Методы работы с последовательностями можно разделить на группы:

* Методы, возвращающие другие последовательности, обработка которых может быть продолжена (filter, select, distinct, sort, group). Все эти методы реализованы lazy-итерируемыми, что позволяет избежать ненужной обработки в случае условного завершения, не доходя до конца перебираемой последовательности.
* Методы принудительной материализации с преобразованием к конкретному виду последовательности (toList, toArray, toMap).
* Методы, извлекающие единственный объект из последовательности (first, firstOrNull).
* Методы-агрегаты, вычисляющие некоторую статистику из последовательности (count, sum, min, max, aggregate).
* Метод-экзекутор, выполняющий некоторое действие, не возвращающее значения на каждом элементе последовательности (each).

Вызов любого из методов на отдельном объекте-wrapper-е не меняет его состояния, позволяя, таким образом делать последующие независимые вызовы. Это происходит благодаря тому, что каждый вызов создает собственный экземпляр внутреннего итератора, в котором и находится состояние вызова каждого метода.

### Используемы техники

#### Локальные классы

Классы, определенные внутри метода представляют удобную замену анонимным классам из мира .Net, позволяя не засорять глобальное пространство имен. Особенно удобно их использование в методах Сервисов Виджетов диаграмм, в которых каждый виджет может формировать собственную модель для отображения точки линии диаграммы.