* **Spring :**
  + **Core**

Центральное положение в фреймворке Spring занимает *контейнер,* управляющий процессом создания и настройки компонентов приложения. Этот модуль содержит фабрику компонентов, обеспечивающую внедрение зависимостей.

* + **Security**

Работает на основе **AOP**.

Некоторые операции, выполняемые программами, являются общими для большинства приложений. Регистрация событий, обеспечение безопасности и управление транзакциями. Функции, охватывающие несколько точек приложения, в разработке программного обеспечения называются *сквозными.* Как правило, сквозные функции концептуально отделены (но часто встроены) от основной логики приложения. Отделение основной логики от сквозных функций – это именно то, для чего предназначено ***аспектно-ориентированное программирование***(**AOP**).

Аспекты помогают отделить сквозные функции. Сквозные функции могут быть описаны как некоторая функциональность, затрагивающая множество мест в приложении. Обеспечение безопасности, например, является такой сквозной функцией – правила соблюдения безопасности могут затрагивать множество методов в приложении.

**Фреймворк Spring Security** обеспечивает безопасность с двух сторон. Для ограничения доступа и обеспечения безопасности на уровне запросов в Spring Security используются сервлет-фильтры. А для обеспечения безопасности на уровне вызовов методов с использованием Spring AOP фреймворк Spring Security предоставляет объекты-обертки и позволяет применять советы, гарантирующие авторизацию пользователей.

***Аутентификация с использованием базы данных***

приложение хранит информацию о пользователях, включая имена, роли и пароли, в реляционной базе данных, Spring Security обеспечивает внутренние механизмы авторизации и аутентификации (шифруем пароль с помощью Spring-овского: BCryptPasswordEncoder), также реализована возможность проброски токена (в течение всей сессии), с данными об доступных ролях, логине и т.п.

***Завершение сеанса работы*** также происходит под контролем Spring security - он сам настраивает фильтр, который будет закрывать сеанс работы с пользователем.

* + **MVC**
* **Model** (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из **POJO**
* **View** (Отображение, Вид) отвечает за отображение данных **Model**, — как правило, генерируя HTML, которые мы видим в своём браузере.
* **Controller** (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую **Model** и передаёт её для отображения в **View**.

Аннотации из фреймворка Spring MVC обеспечивают возможность использования модели программирования, близкую к POJO, облегчая создание контроллеров, обрабатывающих запросы и простых в тестировании.

***Путь одного запроса через Spring***

***MVC***

Каждый раз, когда пользователь щелкает на ссылке или отправляет форму в веб-браузере, запрос отправляется на работу. Наш запрос работает курьером. Подобно почтальону или курьеру из службы доставки, запрос переносит информацию из одного места в другое. Запрос – весьма занятой парень. С момента, когда он покинет браузер, и до момента, когда вернется ответ, запрос сделает несколько остановок, каждый раз сбрасывая часть информации и подбирая что-то взамен. Когда запрос покидает браузер, он несет в себе информацию о требовании пользователя. По крайней мере, запрос будет нести в себе запрошенный **URL**. Но он может также нести дополнительные данные, такие как информация из формы, заполненной пользователем. Первой остановкой на пути запроса является **DispatcherServlet**. Как и большинство веб-фреймворков на языке Java, фреймворк **Spring MVC** пропускает все входящие запросы через единственный сервлет входного контроллера. ***Входной контроллер***(**front controller**) является типичным шаблоном проектирования веб-приложений, где единственный сервлет берет на себя ответственность за передачу всех запросов остальным компонентам приложения, выполняющим

фактическую их обработку. В **Spring MVC** входным контроллером является **DispatcherServlet**.

Задача контроллера **DispatcherServlet** состоит в том, чтобы передать запрос контроллеру **Spring MVC**. Контроллер – это компонент Spring, обрабатывающий запрос. Но приложение может иметь не сколько контроллеров, и входному контроллеру **DispatcherServlet** требуется помощь, чтобы определить, какому контроллеру передать запрос. Поэтому контроллер **DispatcherServlet** консультируется c одним или несколькими механизмами отображения и выясняет, где будет следующая остановка запроса. При принятии решения механизм отображения в первую очередь руководствуется адресом **URL** в запросе. Как только будет выбран соответствующий контроллер, **DispatcherServlet** отправляет запрос в путь к выбранному контроллеру. Достигнув контроллера, запрос отдаст часть своего груза (информацию, отправленную пользователем) и терпеливо будет ждать, пока контроллер обработает эту информацию. (На самом деле хорошо спроектированный контроллер сам почти не занимается обработкой информации, вместо этого он делегирует ответственность за обработку одному или нескольким служебным объектам.) В результате работы контроллера часто появляется некоторая информация, которая должна быть передана назад пользователю и отображена в браузере. Эта информация называется ***моделью***. Но отправки обратно необработанной информации недостаточно, перед отправкой ее следует представить в удобном для пользователя формате, обычно в **HTML**. Для этого информация должна быть передана в одно из ***представлений*** (в нашем случае **JSP**). Последнее, что должен сделать контроллер, – упаковать вместе **модель** и **имя** представления для отображения результатов в браузере. Затем он отсылает запрос вместе с **моделью** и **именем** представления обратно входному контроллеру **DispatcherServlet**. Чтобы контроллер не оказался тесно связанным с каким-либо конкретным представлением, имя представления, возвращаемое входному контроллеру **DispatcherServlet**, не определяет **JSP** страницу непосредственно. Фактически оно даже не предполагает, что представление вообще является страницей **JSP**. Оно является лишь логическим именем представления, используемым затем для поиска фактического представления. Чтобы отобразить логическое имя представления в ссылку на конкретную реализацию, входной контроллер **DispatcherServlet** обратится к арбитру представлений (**View resolver**).

Теперь, когда контроллер **DispatcherServlet** определил, какое представление будет отображать результаты, работа запроса подошла к концу. Его конечная остановка – реализация представления (возможно, страница **JSP**), куда он доставит модель данных. На этом работа запроса заканчивается. На основе модели данных представление создаст отображение страницы, которое будет отправлено обратно клиенту с другим курьером – объектом ответа.

**JSP:**

**Java Server Pages** (**JSP**) обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, так как дизайнерам нет необходимости знать, как работать с динамическими данными. Процессы, выполняемые с файлом JSP при первом вызове:

* Браузер делает запрос к странице JSP.
* **JSP-engine** анализирует содержание файла JSP.
* **JSP-engine** создает временный сервлет с кодом, основанным на исходном тексте файла JSP, при этом контейнер транслирует операторы Java в метод **\_jspService()**. Если нет ошибок компиляции, то этот метод вызывается для непосредственной обработки запроса. Полученный сервлет ответствен за исполнение статических элементов **JSP**, определенных во время разработки в дополнение к созданию динамических элементов.
* Полученный код компилируется в файл **\*.class**.
* Вызываются методы **init()** и **\_jspService()**, и сервлет логически исполняется.
* Сервлет на основе JSP установлен. Комбинация статического HTML и графики вместе с динамическими элементами, определенными в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **ServletResponse**.
* Последующие обращения к файлу JSP просто вызовут метод **\_jspService()** сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен и сервлет не будет выгружен вручную либо пока не будет изменен файл JSP. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности в его **\_jspService()**-метод.
  + **Transaction**

При записи данных необходимо гарантировать целостность данных, выполняя запись в рамках транзакции. Если все операции выполнились успешно, вся транзакция завершается успехом. Иначе операции должны быть отменены, как если бы они и не выполнялись.

Транзакции играют важную роль в программном обеспечении, гарантируя, что данные и ресурсы всегда будут оставаться в непротиворечивом состоянии. Без них было бы можно повредить данные или оставить их в состоянии, противоречащем логике работы приложения.

* ***Atomic*** *(****атомарность****)* – транзакции состоят из одной или более операций, объединенных в единицу работы. Атомарность гарантирует, что-либо будут выполнены все операции, либо ни одна из них. Если все операции выполняются успешно, транзакция завершается успехом. Если какая-то операция терпит неудачу, вся транзакция терпит неудачу и отменяется.
* ***Consistent*** *(****непротиворечивость****)* – после выполнения транзакции (независимо от успеха или неудачи) система остается в состоянии, не противоречащем бизнес-модели. Данные не должны повреждаться.
* ***Isolated*** *(****изолированность****)* – транзакции должны позволять нескольким пользователям работать с одними и теми же данными, не мешая друг другу. То есть транзакции должны быть изолированы друг от друга, предотвращая возможность одновременного чтения и записи одних и тех же данных. (Обратите внимание, что изолированность обычно связана с блокировкой строк и/или таблиц в базе данных.)
* ***Durable*** *(****долговечность****)* – после выполнения транзакции результаты ее выполнения должны сохраняться, чтобы они не терялись в случае ошибок во время работы системы. Под долговечностью обычно понимается сохранение результатов в базе данных или каком-то другом хранилище.
* **Интеграция Spring and JPA Hibernate**

Общее название для такой функциональности – ***объектно-реляционное отображение***(**Object Relational Mapping, ORM**).

Mapping свойств объектов в столбцы базы данных, и SQL-запросы создаются автоматически, без необходимости вручную вводить бесконечные строки из вопросительных знаков.

**Hibernate** – это открытый фреймворк ORM, получивший широкую популярность в сообществе разработчиков. Он обеспечивает не только базовые возможности объектно-реляционного отображения, но также и все прочие «фишки», которые принято ожидать от полнофункционального инструмента ORM:

* ***Отложенная загрузка*** (**lazy loading**) – иерархии наших объектов становятся все более сложными, поэтому иногда бывает нежелательно извлекать все дерево наследования немедленно. Например, приложение выбирает коллекцию объектов **Client**, где каждый из этих объектов, в свою очередь, содержит объекты **Data**, **Login**, и коллекцию объектов **Account**, которые в свою очередь содержат коллекцию **Story**. Если в данный момент приложению требуется получить только атрибуты объектов **Client**, нет смысла извлекать данные для объектов **Story**. Это весьма накладно. Отложенная загрузка позволяет извлекать только те данные, которые действительно необходимы.
* ***Полная загрузка***(**eager fetching**) – полная противоположность отложенной загрузке. Полная загрузка позволяет извлечь все дерево объектов в одном запросе. Когда точно известно, что приложению потребуются все объекты **Client** и в связанные с ними объекты **Data** и **Login**, полная загрузка позволит получить их из базы данных за одну операцию, избавляя от накладных расходов на повторные обращения к ней.
* ***Каскадирование***(**cascading**) – иногда изменения в одной таблице базы данных должны привести к изменениям в других таблицах. Возвращаясь к примеру, с объектами **Client**: при удалении объекта **Client** обычно бывает желательно удалить из базы данных все связанные с ним объекты **Data**, **Login**, **Account**, **Story**.

**Java Persistence API (JPA).**

JPA представляет собой механизм хранения данных на основе POJO,

* **Apache Tomcat**

Это **контейнер**, который позволяет вам использовать интернет приложения такие, как Java сервлеты и **JSP**. Реализует спецификацию сервлетов и спецификацию **Java Server** **Pages** (**JSP**) и **Java Server** **Faces** (**JSF**). Написан на языке Java. **Tomcat** используется в качестве самостоятельного веб-сервера, в качестве сервера контента в сочетании с веб-сервером **Apache** **HTTP Server**, а также в качестве контейнера сервлетов в серверах приложений **JBoss** и **GlassFish**.

**Контейнер**

* *Контейнер клиентского приложения (Application Client Container, ACC) -* набор классов, библиотек и других файлов, необходимых для реализации в приложениях Java SE возможностей Java EE (внедрение зависимостей, управление безопасностью, доступ к ресурсам и т.д.)
* Контейнер ACC обращается к EJB-контейнеру, используя протокол RMI-IIOP, а к веб-контейнеру — по протоколу HTTP (например, для веб-служб).
* Главное назначение контейнера ACC – обеспечить подключение приложения SE к удаленному серверу EE, чтобы открыть доступ к его ресурсам.
* Внутри ACC можно запустить любой Java-класс, обладающий методом main()

Пример одного из способов запуска SE приложения в Glassfish ACC

* Запустить *package-appclient* скрипт собирает все необходимые библиотеки, скрипты и конфигурационные файлы, необходимые для запуска ACC на клиентской машине и упаковывает в *lib/appclient.jar*
* Скопировать и распаковать *appclient.jar* на клиентскую машину
* Запустить клиентское SE приложение в ACC:

*<install-dir>appclient\glassfish\bin\appclient –targetserver <host:port> –jar <path\_to\_client\_jar>*

**Процедура запуска ACC зависит от реализации EE сервера**

* **Logging:**
  + **Slf4j interface**
  + **Logback**

Что нужно уловить в логировании:

1. Уровни логирования. Где ставить error, где trace, где остальное;
2. Цена вызова. Логирование — штука не бесплатная, стоит каких-то милли/нано секунд.
3. Как выключать/включать разные уровни логирования;

Обратите внимание на возможные анти-паттерны в этой области:

* Log and Throw
* Withhold the exception
* Use of System.out.println, System.err.println, e.printStackTrace
* Not using an abstraction layer
* **Unit Tests:**
  + **Junit**
  + **Mockito**

#### Модульное тестирование

Целью *модульного тестирования* является проверка работы прикладной логики всего приложения или отдельных его частей при разных исходных данных, и анализ правильности получаемых результатов. Несмотря на то, что цель **модульного тестирования** выглядит простой и понятной, реализация этого типа тестирования может оказаться очень сложным и запутанным делом, особенно при наличии «старого» кода. Основные приемы проведения модульного тестирования опираются на следующие базовые принципы:

* внешние ресурсы не используются, т.е. недопустимо подключение к базам данных, веб-службам и т.п.;
* каждый класс имеет свой тест;
* тестируются только общедоступные методы или интерфейсы, а внутренний код тестируется за счет изменения входных данных;
* для получения данных, требуемых тестируемой логике, должны создаваться фиктивные зависимости.

При проведении модульного тестирования для создания фиктивных классов-зависимостей можно использовать простой, но мощный фреймворк **Mockito** совместно с [JUnit](http://java-online.ru/blog-junit.xhtml).

## Фреймворк Mockito

Фреймворк *Mockito* предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit тестов.

Наибольшее распространение получили следующие возможности *Mockito*:

* проверка вызова метода и значений передаваемых методу параметров;
* использование концепции «частичной заглушки», при которой заглушка создается на класс с определением поведения, требуемое для некоторых методов класса;
* подключение к реальному классу «шпиона» *spy* для контроля вызова методов.

### Определение поведения - when(mock).thenReturn(value)

Этот метод позволяет определить возвращаемое значение при вызове метода ***mock*** с заданными параметрами. Если будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернётся последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Таким образом, чтобы метод всегда возвращал одно и то же значение, следует просто определить одно условие.

Метод *verify* позволяет проверить, была ли выполнена проверка с определенными параметрами. Если проверка не выполнялась или выполнялась с другими параметрами, то ***verify*** вызовет исключение.

### Подсчет количества вызовов - atLeast, atLeastOnce, atMost, times, never

Для проверки количества вызовов определенных методов *Mockito* предоставляет следующие методы:

* **atLeast (int min)** - не меньше min вызовов;
* **atLeastOnce ()** - хотя бы один вызов;
* **atMost (int max)** - не более max вызовов;
* **times (int cnt)** - cnt вызовов;
* **never ()** - вызовов не было;

**Использование шпиона spy на реальных объектах**

Mockito позволяет подключать к реальным объектам «шпиона» spy, контролировать возвращаемые методами значения и отслеживать количество вызовов методов