**Система сборки** – это инструмент, преобразующий исходный код в программу, которую можно запускать. Существует несколько систем сборок:

* ***Apache Ant*** – использует императивный подход с составлением алгоритма действий в *XML*-файле.
* ***Apache Maven*** – использует декларативный подход с описанием конечного результата.
* ***Gradle*** – комбинирует императивный (сценарии сборки) и декларативный (дерево зависимостей) подходы.

Преимущества *Maven*:

* Декларативное описание проекта. Для этого используется файл *pom.xml*, где объявляется, что должно получиться в результате (не требует конкретных команд).
* Автоматическое управление зависимостями. Система сама скачивает зависимости и разрешает конфликты версий.
* Фиксированные циклы сборки проекта, состоящие из определенной последовательности фаз, с достижением конкретных целей в каждой.
* Четкая структура каталогов, где располагаются исходный код, тесты и другие ресурсы.
* Модульная структура, позволяющая легко расширить функциональность проекта.

**Архетип** – это шаблон, который можно использовать в качестве основы для своего проекта. Он помогает стандартизировать разработку и получить доступ к уже готовым решениям.

**Артефакт** – это то, что создано в процессе разработки программного обеспечения. Им может быть любая библиотека, зависимость или плагин.

**Артефакт *Maven*** – это файл *JAR* формата, который появляется при сборке проекта. У него есть своя группа настроек:

* ***groupId*** – уникальный идентификатор группы, которые создали проект. Как правило он основан на полном доменном имени организации (например, *ru.yandex.practicum*).
* ***artifactId*** – уникальное имя артефакта, генерируемого этим проектом.
* ***version*** – версия артефакта.

Ссылка на гайд по основам *XML* - <https://doka.guide/tools/xml/>

Для оформления *pom.xml* также могут использоваться следующие **теги**:

* *<build> </build>* - блок настройки процесса сборки
* *<plugins> </plugins>* - блок настройки плагинов
* *<resources> </ resources>* - блок настройки доп. ресурсов
* *<dependencies> </ dependencies>* - блок настройки зависимостей
* *<dependencyManagement> </ dependencyManagement>* - блок настройки зависимостей в *parent-POM*
* *<goals> </ goals>* - блок настройки задач/целей
* *<profiles> </ profiles>* - блок настройки профилей создания
* *<developers> </ developers>* - блок настройки разработчиков
* *<mailingLists> </ mailingLists>* - блок настройки списков рассылки
* *<parent> </ parent>* - блок настройки родительского проекта
* *<scope> </scope>* - модификатор, указывает на каком этапе сборки используется данный артефакт

Зависимости нужны для того, чтобы работали аннотации. Плагины нужны для изменения процесса компиляции/сборки в *Lifecycle*.

Ссылка на документацию Maven - <https://maven.apache.org/pom.html#Maven_Coordinates>

**Репозиторий *Maven*** – это директория, где хранятся все файлы проекта, библиотеки, плагины и любые другие артефакты. Существуют локальный репозиторий (папка с расширением *.m2*), удаленные репозитории сообществ (через протоколы *https://*, *file://* и другие) и центральный репозиторий Maven Central (<https://repo.maven.apache.org/maven2/>). Инструкции по подключению можно найти на <https://search.maven.org/> и <https://mvnrepository.com/>. Пример подключения:

*<repositories>*

*<repository>*

*<id>additional-local-dir</id>*

*<url>file://path/to/directory</url>*

*</repository>*

*<repository>*

*<id>org-remote-repository</id>*

*<url>https://maindomain.org/maven2</url>*

*</repository>*

*<repositories>*

Приоритеты поиска зависимостей в репозиториях локальный -> центральный -> удаленный.

Процесс сборки включает несколько жизненных циклов. Всего их три:

1. ***clean*** – в ходе этого цикла происходит очистка проекта.
2. ***default*** – основной цикл сборки. В ходе него происходит компиляция кода, генерация тестов, сборка дополнительных артефактов, различного рода проверки и т.д. Включает 23 фазы.
3. ***site*** – посвящен созданию проектной документации.

Каждый цикл включает набор фаз. Есть подготовительные (*pre-*), основные и завершающие (*post-*) фазы. Ссылка на фазы циклов - <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>.

Следующий плагин с помощью команды ***surefire-report:report***создает HTML-страницу с отчетом о пройденных тестах в удобном для чтения виде.

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-surefire-report-plugin</artifactId>*

*<version>2.22.2</version>*

*</plugin>*

**Дерево зависимостей** (***dependency tree***) – совокупность зависимостей всех компонентов, используемых в проекте. Посмотреть все зависимости можно с помощью команды ***dependency:tree***. Зависимости, которые появились в проекте неявно, называются **транзитивными** (***transitive***) или переходными.

Когда для работы пакетов требуются разные версии одной и той же библиотеки, проект попадает в **ад зависимостей** (***dependency hell***). Из-за этого проект может не запуститься. В качестве решения предлагается файл ***BOM*** (***Bill of Material***) или **список материалов**.

В *maven* реализована возможность **наследования** от *parent*-проекта с помощью тега *<parent></parent>*. Для создания проекта-предка используется строка *<packaging>pom</packaging>*. Изменения в проекте-наследнике имеют приоритет над проектом-предком. Благодаря наследования можно переиспользовать все связанные зависимости. Файл *BOM* является предком для *POM*-файла. Например, с помощью наследования *BOM*-файла можно указать все версии зависимостей и не указывать их (или переопределить версию нужной зависимости в блоке *<properties>*) в *POM*-файле.

При наследовании может быть только один проект-предок и нельзя подключить второй *BOM*-файл. Обойти это ограничение можно, подключив *BOM* как зависимость с использованием строк *<type>pom</type>* и *<scope>import</scope>*.

<https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm#MAVEN8855>

<https://medium.com/javarevisited/10-important-maven-plugins-for-java-developers-330b98b71720>

<https://habr.com/ru/post/205118/>

<https://github.com/eirslett/frontend-maven-plugin>

<https://www.baeldung.com/maven-polyglot>

Гайды на postman <https://habr.com/ru/company/maxilect/blog/596789/>

<https://learning.postman.com/docs/getting-started/introduction/>

**Лог** (***log***) или **журнал регистрации** – это набор записей о происходящих в программе событиях (ошибках, действиях пользователя и т.д.). Процесс добавления в журнал новых записей называется **логированием**. У каждой записи есть точная метка даты и времени. Из записей лога можно:

* Получить информацию о характеристиках окружения, в котором запущена программа.
* Разобраться с ошибками.
* Устранить проблемы с производительностью.
* Обнаружить подозрительное поведение пользователя и предотвратить злоумышленные действия.
* Предотвратить проблемы с инфраструктурой.

Большинство известных фреймворков логирования в *Java* состоят из трех частей:

1. ***Logger*** (**журнал**) – специальный класс, который преобразует событие в объект и передает его дальше другим компонентам фреймворка.
2. ***Appender*** (**добавлять**) – интерфейс, который отвечает за добавление объекта, полученного от логера, непосредственно в лог. В зависимости от местоположения журнала используются разные реализации. Запись всегда добавляется в конец лога.
3. ***Layout*** (**формат**) – интерфейс, который перед выводом записи из лога в консоль или на печать приводит ее к определенному формату.

В качестве интерфейса единого *API* логирования мы будем использовать фреймворк ***The Simple Logging Façade for Java*** ([***SLF4J***](https://www.slf4j.org/)), а в качестве реализации *API* – фреймворк [***Logback***](https://logback.qos.ch/).

1. Для получения нового объекта класса ***Logger*** используется класс-фабрика ***LoggerFactory*** и его статический метод ***getLogger(String)***. Во время работы программы нет смысла менять экземпляр логера, поэтому можно пометить переменную как final. В отличие от констант, переменную логера принято записывать в нижнемВерблюжьемРегистре.

*private static final Logger log = LoggerFactory.getLogger("Practicum");*

1. В составе *Logback* есть большой набор готовых реализаций типа *Appender*:

* ***ConsoleAppender*** – выводит записи в консоль.
* ***FileAppender*** – сохраняет записи в файл.
* ***RollerFileAppender*** – сохраняет записи в файл, но при этом умеет автоматически создавать новый файл при достижении заданного лимита размера. Лимит задается программистом.

1. В большинстве случаев в *Logback* для интерфейса *Layout* используется реализация по умолчанию ***PatternLayout***. Например, шаблон %d{HH:mm:ss}: %message%n выведет лог в таком виде:

*14:06:49: Событие 1*

*14:27:53: Событие 2 // и так далее*

В *IDEA* по умолчанию вывод лога проекта перенаправляется на консоль с помощью компонента *ConsoleAppender*. Также есть режим ***soft wraps*** — перенос текста на новую строку, если она не помещается в видимую область окна консоли.

Для удобства работы с большими объемами записей в *SLF4J* были созданы пять **специальных уровней важности** и соответствующие им методы ***trace***, ***debug***, ***info***, ***warn***, ***error***:

* ***TRACE*** (**отслеживать**) – для трассировочных сообщений, которые предоставляют очень подробную информацию о каком-либо процессе. На этом уровне может записываться все что происходит в системе (выбор ветки условий, значений параметров внутри циклов, каждый шаг алгоритма и т.д.).
* ***DEBUG*** (**отлаживать**) – для отладочных сообщений, которые важны во время анализа некорректного поведения программы. Чаще всего в таких записях содержится информация о значениях переменных, которые относятся к бизнес-логике приложения.
* ***INFO*** (**информация**) – для сообщений, которые помогают понять текущее состояние программы. Например, информация о статусе обработки какого-либо запроса, результате авторизации пользователя, сведения об *IP*-адресе или сетевом порте, на котором запустилось веб-приложение.
* ***WARN*** (**предупреждать**) – для сообщений с предупреждениями, которые появляются в случае, если что-то пошло не так. Например, было сгенерировано исключение или обнаружена проблема в ходе выполнения алгоритма (допустим переданы данные в неверном формате). Программа может обработать эту ситуацию самостоятельно и продолжить работу.
* ***ERROR*** (**ошибка**) – для сообщений о критических ошибках, требующих оперативного вмешательства, без которого программа не сможет продолжить работу. Например, нехватка ресурсов, невозможность установить соединение с базой данных и т.д.

В процессе логирования можно использовать разные комбинации входных параметров:

* ***String msg*** – логируется только переданное сообщение
* ***String format, Object… arguments*** – логируется переданное сообщение и набор аргументов.

*log.info("Пользователь {} сменил статус на {}.", "Григорий Логинов", "Занят");*

* ***String msg, Throwable t*** – логируется переданное сообщение и полученное исключение. Полезно в блоке *try-catch*.

По умолчанию логируются все сообщения, кроме трассировочных. Однако это можно изменить, указав уровень важности, начиная с которого сообщения будут логироваться. Также можно выключить логирование с помощью уровня ***OFF***. Иерархия уровней выглядит так (от наименее важного к наиболее важному):

*TRACE < DEBUG < INFO < WARN < ERROR*

Для изменения уровня логирования экземпляр класса *Logger* нужно привести к типу логера конкретной системы (в *Lomback* это класс ***ch.qos.logback.classic.Logger***), в котором есть метод ***setLevel(Level newLevel)***. Это можно также сделать в конфигурационном файле *application.properties* через добавление строки:

*logging.level.полное\_имя\_логгера=уровень\_важности*

*logging.level.ru.yandex.practicum=debug // при указании корневого пакета, изменятся уровни всех логеров*

**Стартер** (***starter***) – это набор библиотек и конфигураций, которые специально создается для транзитивного внесения в проект нужных зависимостей. Например, в стартере *spring-boot-starter-web* транзитивно подключается фреймворк логирования *spring-boot-starter-logging*.

Любой стартер от команды *Spring Framework* транзитивно добавляет в проект фреймворки логирования ***logback-core*** и ***slf4j-api***, а в Spring Boot есть специальный стартер с ними *spring-boot-starter-logging*.

Добавить информацию для логирования можно несколькими способами:

1. **Вручную**. Из минусов – много информации придется постоянно обновлять в ручном режиме.

log.info(“Информация для логирования с уровнем INFO.”);

1. **Средствами *Spring Framework***. Требуется изменить уровень логера сервера запросов в конфигурационном файле *application.properties*:

*logging.level.org.springframework.web.servlet.DispatcherServlet=debug*

1. **Через стороннюю библиотеку**. Например, подключить зависимость *LogBook* (логирует в *JSON*), и изменить уровень логирования запросов этим пакетом в конфигурационном файле *application.properties*:

*<dependency>*

*<groupId>org.zalando</groupId>*

*<artifactId>logbook-spring-boot-starter</artifactId>*

*<version>2.14.0</version>*

*</dependency>*

*logging.level.org.zalando.logbook=TRACE*

**Библиотека** [***Lombok***](https://projectlombok.org/) позволяет автоматически генерировать геттеры, сеттеры, конструкторы, методы *toString()*, *equals()* и *hashCode()*, а также автоматизировать создание переменных для логирования с помощью аннотации и многое другое. Подключить зависимость можно так:

*<dependency>*

*<groupId>org.projectlombok</groupId>*

*<artifactId>lombok</artifactId>*

*<version>1.18.20</version>*

*<scope>provided</scope>*

*</dependency>*

*Lombok* используется только на этапе компиляции программы. На это указывает тег ***provided***.

[Полный список аннотаций *Lombok*](https://projectlombok.org/features/all)

Аннотации ***@Getter*** и ***@Setter*** перед переменной/классом позволяют автоматически генерировать соответствующие методы. Названия сгенерированных методов будут звучать как ***getИмяПеременной*** и ***setИмяПеременной***.

Аннотация ***@ToString*** перед классом генерирует реализацию метода *toString()*. В генерируемый метод будут добавлены имя класса, названия всех полей и их значений.

Аннотация ***@EqualsAndHashCode*** перед классом генерирует реализации методов *equals()* и *hashCode()*. В генерируемые методы будут добавлены все нестатические поля класса. Чтобы добавить или исключить из генерируемой реализации какие-либо поля, нужно пометить их аннотациями ***@EqualsAndHashCode.Include*** и ***@EqualsAndHashCode.Exclude*** соответственно (при условии отсутствия аннотации на классе).

Аннотация ***@NoArgsConstructor*** генерирует конструктор без параметров. Использование с флагом ***force=true*** установит значения по умолчанию для этих полей.

Аннотация ***@AllArgsConstructor*** генерирует конструктор, включающий все возможные поля.

Аннотация ***@RequiredArgsConstructor*** генерирует конструктор, включающий все *final*-поля или поля, помеченный аннотацией ***@NonNull***.

Аннотация ***@NonNull*** генерирует проверку поля на неинициализированное значение *null*. При проверке возможна генерация исключения *NullPointerException*.

Аннотация ***@Data*** является группировкой сразу пяти аннотаций - *@Getter*, *@Setter*, *@ToString*, *@EqualsAndHashCode* и *@RequiredArgsConstructor*. Она создает полностью **изменяемый** (***muttable***) объект.

Аннотация ***@Value*** является группировкой сразу четырех аннотаций - *@Getter*, *@ToString*, *@EqualsAndHashCode* и *@AllArgsConstructor*. Она создает полностью **неизменяемый** (***immuttable***) объект с *final* и *private* полями.

Аннотация ***@Slf4j*** генерирует объявление переменной *log* и подключение необходимых для нее импортов библиотек логирования *slf4j*.

Библиотека ***MapStruct*** позволяет автоматически генерировать методы преобразования одного объекта в другой (мапперы).

Сначала нужно подключить в проект зависимости:

*<dependency>*

*<groupId>org.mapstruct</groupId>*

*<artifactId>mapstruct</artifactId>*

*<version>1.5.3.Final</version>*

*</dependency>*

*<dependency>*

*<groupId>org.mapstruct</groupId>*

*<artifactId>mapstruct-processor</artifactId>*

*<version>1.5.3.Final</version>*

*</dependency>*

Затем создать интерфейс с аннотацией ***@Mapper***, в котором будут находиться специально именованные методы. Для использования в *Spring* или *CDI* дополнительно указывается параметр со значением *spring* или *cdi*. Затем добавить маппер в нужный класс.

*@Mapper(componentModel = “spring”)*

*public interface SourceDestinationMapper {*

*Destination sourceToDestination(Source source);*

*Source destinationToSource(Destination destination); }*

Аннотация ***@Mapping*** позволяет сопоставить исходное и целевое поле при маппинге. Указывается перед методом в любом количестве.

*@Mapper*

*public interface UserMapper {*

*@Mapping(target=”userId”, source=”entity.id”)*

*@Mapping(target=”name”, source=”entity.name”)*

*UserDto userToUserDto(User entity); }*

Также библиотека может преобразовывать объекты с конвертацией полей (например, *String* и *Date*). Подробнее о работе библиотеки можно почитать [тут](https://www.baeldung.com/mapstruct).