1. **Платформа Java EE. Спецификации и их реализации.**

Изображение выглядит как текст, табло

Автоматически созданное описание

Java EE — это платформа, построенная на основе Java SE, которая предоставляет API и среду времени выполнения для разработки и запуска крупномасштабных, многоуровневых, масштабируемых, надежных и безопасных сетевых приложений.

Подобные приложения называют корпоративными (Enterprise applications), так как они решают проблемы, с которыми сталкиваются большие бизнесы.

Java EE представляет из себя набор спецификаций и документации, описывающий архитектуру серверной платформы для задач средних и крупных предприятий. Сервер приложений Java EE (часто называемый контейнером) — это реализация системы в соответствии со спецификацией, обеспечивающая работу модулей с логикой конкретного приложения.

Деление на платформы (Java SE, Java EE, Java Card ...) появилось в Java EE 6 и позволяет сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE. Существует только 2 профиля — Full и Web. Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля.

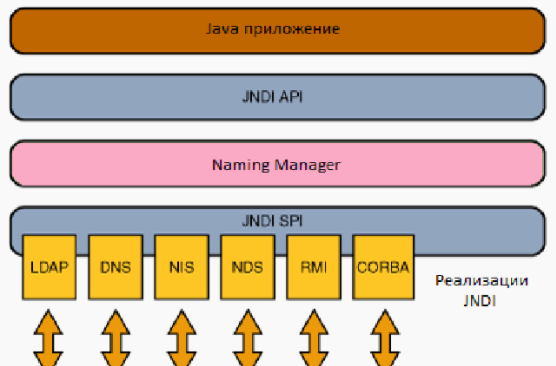
1. **Принципы IoC, CDI и Location Transpanency. Компоненты и контейнеры.**

IoC - Inversion of Control (применительно к Java EE):

* Жизненным циклом компонента управляет контейнер (а не программист).
* Объекты создает не программист (используя new), а контейнер IoC. Применяется далеко не ко всем объектам в приложении, а только к управляемым (в Spring это классы с аннотациями @Component, @Service и т.д., в EJB — бобы @Stateless, @Stateful, @MessageDriven).
* За взаимодействие между компонентами отвечает тоже контейнер.

CDI — Contexts and Dependency Injection (позволяет снизить (или совсем убрать) зависимость компонента от контейнера):как бы компонент и так зависит от контейнера, так как тот его порождает, но тут о другом: программист, когда пишет компонент, он не зависит от API, предоставляемых контейнером

* Не требуется реализации каких-либо интерфейсов.
* Не нужны прямые вызовы API.
* Реализуется через аннотации.

Оба этиx подхода реализованы с помощью JNDI

JNDI - набор API, позволяющий доставать объекты из контейнера по их именам. По сути, это замена обращению по ссылкам, и благодаря этому легко реализуется Dependency Lookup и Dependency Injection. Хранение объектов в таком случае можно представить как Map, где ключи - это имена, а значения - нужные объекты (например, ссылки на них).

Пример использования JNDI

public void jndi() {

try {

// Инициализируем контекст по умолчанию.

Context context = new InitialContext();

dataSource = (DataSource) context.lookup("Database");

} catch (NamingException e) {

// Ссылка не найдена

}

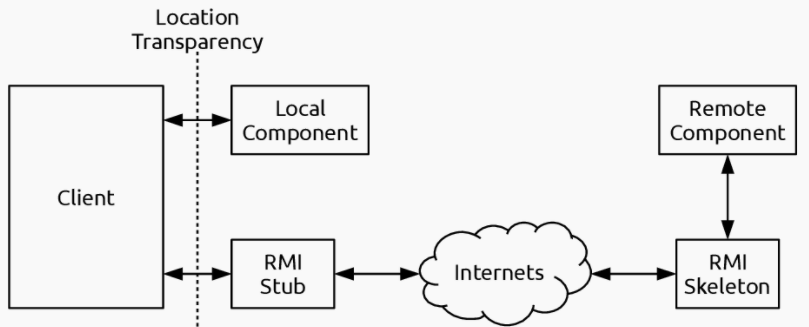
}

**Location Transparency** - прозрачность местонахождения - это что-то вроде паттерна. Используя предыдущие механики, мы подходим к этой.

Инфраструктура завязана на RMI (Java API, позволяющий вызывать методы удаленных объектов)

Благодаря CDI не важно, где физически расположен вызываемый компонент — за его вызов отвечает контейнер.

Клиент имеет прокси, обращается с ней как с нужным объектом. Прокси перенаправляет его вызовы куда следует: если нужный компонент находится локально, идёт к нему обращение по ссылке; если удалённо (в другой JVM) - необходима сериализация передаваемых данных.



Компоненты и контейнеры

* Приложение состоит из компонентов и контейнера, управляющего жизненным циклом компонентов. Пример: Servlet - компонент, Glassfish - контейнер.
* Голая Java EE — это просто набор спецификаций.
* Данные спецификации реализуют различные Java EE сервера.
* Java EE сервер — это серверное приложение, которое реализует API-интерфейсы платформы Java EE и предоставляет стандартные службы Java EE. Серверы Java EE иногда называют серверами приложений. Данные сервера могут содержать в себе компоненты приложения, каждый из которых соответствует своему уровню в многоуровневой иерархии. Сервер Java EE предоставляет этим компонентам различные сервисы в форме контейнера.
* Контейнеры — это интерфейс между размещенными на них компонентами и низкоуровневыми платформо-независимыми функциональными возможностями, поддерживающими компонент.
* Контейнеры предоставляют размещенным на них компонентам определенные службы. Например, управление жизненным циклом разработки, внедрение зависимости, параллельный доступ и т. д. Контейнеры скрывают техническую сложность и повышают мобильность.

1. **Управление жизненным циклом компонентов. Дескрипторы развёртывания.**

По большей части, жизненным циклом компонентов управляет контейнер. Тем не менее, есть возможность влиять на этот процесс с помощью аннотаций / xml / прочих файлов настроек. Дескрипторы - это как раз такие файлы (в том числе xml). В них можно задавать свойства используемых в приложении компонентов: имена, ссылки, параметры. Также можно настраивать то, как итоговое приложение будет собираться и как оно будет взаимодействовать с внешним миром.

Дескриптор развёртывания - конфигурационный файл артефакта. Дескриптор развёртывания описывает то, как компонент, модуль или приложение (такое, как веб-приложение или приложение предприятия) должно быть развёрнуто

Примеры дескрипторов развёртывания:

* web.xml - дескриптор развёртывания веб-приложений (упаковываемых обычно в .war архивы)
* ejb-jar.xml - дескриптор развёртывания EJB-приложения
* application.xml - дескриптор развёртывания приложения, использующего несколько web.xml / ejb-jar.xml

1. **Java EE API. Виды компонентов. Профили платформы Java EE.**

Деление на платформы (Java SE, Java EE, Java Card ...) появилось в Java EE 6 и позволяет сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE. Существует только 2 профиля — Full и Web. Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля. Пример Web profile: Tomcat Пример Full profile: Glassfish

Виды компонентов:

* веб-компоненты (сервлеты, jsp, ...) - формируют содержимое представления
* Java-бины (POJO, ManagedBean, EJB) - позволяют изменять данные и осуществлять их временное хранение, взаимодействовать с базами данных и веб-службами, а также отображать содержимое в ответ на запросы клиентов.

1. **Компоненты EJB. Stateless & Stateful Session Beans. EJB Lite и EJB Full.**
2. **Работа с электронной почтой в Java EE. JavaMail API.**
3. **JMS. Реализация очередей сообщений. Способы доставки сообщений до клиента. Message-Driven Beans.**
4. **Понятие транзакции. Управление транзакциями в Java EE. JTA.**
5. **Веб-сервисы. Технологии JAX-RS и JAX-WS.**
6. **Платформа Spring. Сходства и отличия с Java EE.**

Несмотря на то, что Spring не обеспечивал какую-либо конкретную модель программирования, он стал широко распространённым в Java-сообществе главным образом как альтернатива и замена модели Enterprise JavaBeans. Spring предоставляет бо́льшую свободу Java-разработчикам в проектировании; кроме того, он предоставляет хорошо документированные и лёгкие в использовании средства решения проблем, возникающих при создании приложений корпоративного масштаба.

Между тем, особенности ядра Spring применимы в любом Java-приложении, и существует множество расширений и усовершенствований для построения веб-приложений на Java Enterprise платформе. По этим причинам Spring приобрёл большую популярность и признаётся разработчиками как стратегически важный фреймворк.

Пожалуй, наиболее трудной частью объяснения технологии Spring является точная классификация того, что она собой представляет. Обычно Spring описывают как облегченную платформу для построения Jаvа-приложений, но с этим утверждением связаны два интересных момента.

Во-первых, Spring можно применять для построения любого приложения на языке Java (например, автономных, веб-приложений или приложений Java Enterprise Edition (JEE)) в отличие от многих других платформ (таких как Apache Struts, которая ограничена созданием только веб-приложений).

Во-вторых, характеристика облегченная в действительности не имеет никакого отношения к количеству классов или размеру дистрибутива, а вместо этого определяет принцип всей философии Spring — минимальное воздействие. Платформа Spring является облегченной в том смысле, что для использования всех преимуществ ядра Spring вы должны вносить минимальные (если вообще какие-либо) изменения в код своего приложения, а если в какой-то момент вы решите прекратить пользоваться Spring, то и это сделать очень просто.

Обратите внимание, что речь идет только о ядре Spring - многие дополнительные компоненты Spring, такие как доступ к данным, требуют более тесной привязки к Spring Framework. Однако польза от такой привязки вполне очевидна.

Spring, вероятно, наиболее известен как источник расширений (features), нужных для эффективной разработки сложных бизнес-приложений вне тяжеловесных программных моделей, которые исторически были доминирующими в промышленности. Ещё одно его достоинство в том, что он ввел ранее неиспользуемые функциональные возможности в сегодняшние господствующие методы разработки, даже вне платформы Java.

Этот фреймворк предлагает последовательную модель и делает её применимой к большинству типов приложений, которые уже созданы на основе платформы Java. Считается, что Spring реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java.

Преимущества:

Spring Framework обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе Java. Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. Spring Framework не всецело связан с платформой Java Enterprise, несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности.

* Относительная легкость в изучении и применении фреймворка в разработке и поддержке приложения.
* Внедрение зависимостей (DI) и инверсия управления (IoC) позволяют писать независимые друг от друга компоненты, что дает преимущества в командной разработке, переносимости модулей и т.д..
* Spring IoC контейнер управляет жизненным циклом Spring Bean и настраивается наподобие JNDI lookup (поиска).
* Проект Spring содержит в себе множество подпроектов, которые затрагивают важные части создания софта, такие как вебсервисы, веб программирование, работа с базами данных, загрузка файлов, обработка ошибок и многое другое. Всё это настраивается в едином формате и упрощает поддержку приложения.

Отличие Spring от JavaEE

Проще говоря, Java EE — для легко масштабируемого монолитного приложения, Spring — для совсем маленьких приложений с GUI на Front-end или для микросервисной архитектуры.

Конкурентном Java EE считается Spring Framework. Если взглянуть на развитие двух данных платформ, выходит интересная картина. Первые версии Java EE были созданы при участии IBM. Они вышли крутыми, но неповоротливыми, тяжеловесными, неудобными в использовании. Разработчики плевались из-за необходимости поддерживать большое количество конфигурационных файлов и из-за прочих причин, усложняющих разработку.

В то же время на свет появился Spring IoC. Это была маленькая, красивая и приятная в обращении библиотека. В ней также использовался конфигурационный файл, но в отличии от Java EE, он был один. Простота Spring привела к тому, что практически все стали использовать данный фреймворк в своих проектах.

А далее Spring и Java EE начали свой путь к одному и тому же, но с разных концов. Компания Pivotal Software, разработчик Spring, стали выпускать проект за проектом, чтобы покрыть все возможные и невозможные потребности Java-разработчиков. Постепенно то, что раньше называлось Spring, сначала стало одним из проектов, а потом и вовсе слилось с несколькими другими проектами в Spring Core. Все это привело к неминуемому усложнению Spring по сравнению с тем, каким он был изначально. Со временем следить за всем клубком зависимостей спринга стало уж совсем сложно, и возникла потребность в отдельной библиотеке, которая стала бы загружать и запускать все сама (сейчас где-то икнул так горячо любимый Spring Boot).

Все это время JCP работал над одним — добиться максимального упрощения всего, что только можно внутри Java EE. В итоге в современном EJB для описания бина достаточно указать одну аннотацию над классом, что предоставляет разработчику доступ ко всей мощи технологии Enterprise Java Beans. И подобные упрощения затронули каждую спецификацию внутри Java EE.

В итоге по функционалу Spring и Java EE примерно разделяют паритет. Где-то что-то лучше, где-то что-то хуже, но если смотреть глобально, больших различий нет. То же самое касается сложности работы. И Spring, и Java EE являются превосходными инструментами. Пожалуй, лучшими из того, что сейчас есть, для построения корпоративных сетевых приложений на языке Java.

Java EE:

* Java EE промышленности утверждены типовые основы API
* Он преимущественно основан на аннотациях и CDI
* JFC MVC framework для веб-разработки
* JPA спецификация для обработки операции DB
* JTA API с внедрением
* EJB контейнерная и POJO основанная реализация
* Oracle лицензия

Spring:

* На основе IOC и AOP
* На основе конфигурации XML (теперь они используют аннотацию)
* Использует фреймворк Spring DAO (основанный на шаблоне дизайна шаблона) для подключения к базе данных
* Предоставляет уровень абстракции для поддержки различных реализаций JTA.
* Интегрируется с различными поставщиками Java для поддержки различных возможностей, таких как struts и т. д
* Обеспечивает платформу end-to-end для построения веб-приложения, достигая свободной связи с использованием DI и AOP
* Лицензия с открытым исходным кодом

1. **Модули Spring. Архитектура Spring Runtime. Spring Security и Spring Data.**

Spring потенциально может быть универсальным магазином для всех ваших корпоративных приложений. Однако Spring является модульным, позволяя вам выбирать, какие модули вам подходят, без необходимости вносить остальные. В следующем разделе приведены сведения обо всех модулях, доступных в Spring Framework.

Основной контейнер

Базовый контейнер состоит из модулей Core, Beans, Context и Expression Language, подробности которых следующие:

* Модуль Core обеспечивает основные части платформы, включая функции IoC и Dependency Injection.
* Модуль Bean предоставляет BeanFactory, которая представляет собой сложную реализацию фабричного шаблона.
* Модуль Context основан на прочной основе, предоставляемой модулями Core и Beans, и является средой для доступа к любым объектам, определенным и настроенным. Интерфейс ApplicationContext является координационным центром модуля Context.
* Модуль SpEL предоставляет мощный язык выражений для запросов и манипулирования графом объектов во время выполнения.

Уровень доступа к данным / интеграции состоит из модулей JDBC, ORM, OXM, JMS и Transaction, подробности которых следующие:

* Модуль JDBC предоставляет уровень абстракции JDBC, который устраняет необходимость в утомительном кодировании, связанном с JDBC.
* Модуль ORM предоставляет слои интеграции для популярных API объектно-реляционного отображения, включая JPA, JDO, Hibernate и iBatis.
* Модуль OXM предоставляет уровень абстракции, который поддерживает реализации отображения объектов / XML для JAXB, Castor, XMLBeans, JiBX и XStream.
* Модуль JMS Java Messaging Service содержит функции для создания и потребления сообщений.
* Модуль Transaction поддерживает программное и декларативное управление транзакциями для классов, которые реализуют специальные интерфейсы, и для всех ваших POJO.

Spring Security — это фреймворк, который сфокусирован на обеспечение как аутентификации, так и авторизации в Java-приложениях. Как и все Spring проекты, настоящая сила Spring Security в том, что он может быть легко дополнен нужным функционалом.

* Комплексная и расширяемая поддержка как аутентификации, так и авторизации
* Защита от атак типа фиксация сессии, кликджекинг, межсайтовая подделка запроса и др.
* Интеграция с Servlet API
* Интеграция с Spring Web MVC при необходимости

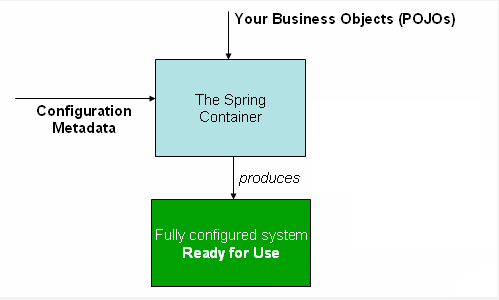
Миссией Spring DATA является предоставление единой модели программирования с использованием Spring для доступа к данным, сохраняя при этом специальные черты базового хранилища.

Фреймворк позволяет облегчить использование технологий доступа к данным, реляционных и не реляционных баз данных, облачных баз данных. Spring DATA — базовый проект, который включает множество других подпроектов, которые работают с конкретными базами данных.

1. **Реализация IoC и CDI в Spring. Сходства и отличия с Java EE.**

Описание работы IoC контейнера

Ниже представлена диаграмма, отражающая, как работает Spring. Ваши классы приложения совмещаются с метаданными конфигурации, в результате чего будет создан и инициализирован ApplicationContext, а на выходе вы получите полностью настроенное и готовое к выполнению приложение.



ApplicationContext представляет собой Spring IoC контейнер и необходим для инициализации, настройки и сборки бинов для построения приложения.

В метаданных конфигурации разработчик описывает как инициализировать, настроить IoC контейнер и собрать объекты в вашем приложении. В данном и других уроках этого цикла везде, где возможно, будет использоваться подход на основе аннотаций и Java-конфигурации. Если вы сторонник XML-конфигурации, либо хотите посмотреть как делать тоже самое через XML, обратитесь к оригинальной документации по Spring Framework или соответствующего модуля/проекта.

Основными признаками и частями Java-конфигурации IoC контейнера являются классы с аннотацией @Configuration и методы с аннотацией @Bean. Аннотация @Bean используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый Spring IoC контейнером. Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component(или её наследниках). Класс с аннотацией @Configuration говорит о том, что он является источником определения бинов. Самая простейшая из возможных конфигураций выглядит следующим образом:

Для того, чтобы приступить к настройке и изучению Spring IoC контейнера, вы должны инициализировать ApplicationContext, который поможет также с разрешением зависимостей. Для обычной Java-конфигурации применяется AnnotationConfigApplicationContext, в качестве аргумента к которому передается класс, либо список классов с аннотацией @Configuration, либо с любой другой аннотацией JSR-330, в том числе и @Component

Отличия

Есть IoC - инверсия контроля (я завишу от кого-то, но не я буду его дергать (хотя это нужно мне), а кто-то "сверху" (контейнер (контекст)) будет поставлять мне то, от чего я хочу зависеть, таким образом достигается меньшая связанность, а => гибкость ).

Ну и IoC это просто концепция (подход), а вот DI (в Spring) и CDI (в Java EE) - это реализации этого подхода. Да и прикол в том, что почти идентичные (по сути делают они одно и то же, но названия аннотаций разные)

Напоминание: Фактически в Java EE стеке изначально и использовался спринговый DI. Но после того как его включили прямо в Core, использовать его отдельно стало невозможно. Пришлось делать свой DI с аннотациями и программистками.

Но почему в Spring DI, а в Java EE CDI? Я могу предположить, что сязано это с тем, что для Java EE обязательно нужен Enterprise Application Server’a (Tomcat таким не является), поэтому Context всегда уже реализован в нем и содержится в этом сервере.

А что касается Spring, там используется DI, так как Context всегда реализуется самим програмистом (ApplicationContext)

1. **Реализация REST API в Java EE и Spring.**
2. **React JS. Архитектура и основные принципы разработки приложений.**
3. **Компоненты React. State & props. "Умные" и "глупые" компоненты.**
4. **Разметка страниц в React-приложениях. JSX.**
5. **Навигация в React-приложениях. ReactRouter.**
6. **Управление состоянием интерфейса. Redux.**
7. **Angular: архитектура и основные принципы разработки приложений.**

Angular - в отличие от react это фреймворк. В общем случае написан на TypeScript. За 4 года 9 релизов.

Angular - развитие библиотеки AngularJS. Т. е в 2016 году Angular=fork AngularJS. Дальше они развивались независимо.Так как Angular - это фреймворк, поэтому, чтобы начать разрабатывать приложение на нем, нужно сделать некую предварительную работу. Какую?

* Для разработки нужно настроить соборочное окружение (на базе node.js и npm). Ну и в этом окружении клиентскую часть нашего приложения собирать
* Приложения состоят из модулей ( наследники класса NgModules)
* Модули обеспечивают контекст для компонентов (components)
* Из компонентов строятся представления (views)
* Компоненты взаимодействуют с сервисами (элементы, реализующие логическую составляющую нашего фронта) (services) с помощью DI

1. **Angular: модули, компоненты, сервисы и DI.**

Модули

* Не совсем то же самое, что модули в ES6 (хотя и похожи)
* Каждый модуль обеспечивает контекст компиляции для одного или группы компонентов
* Модули могут связывать компоненты с сервисами, необходимыми для работы этих компонентов
* Группировка компонентов по модулям - на усмотрение программиста
* Каждое приложение обязательно включает в себя корневой модуль (root module) под названием AppModule (файл app.module.ts)
* Модули могут ссылаться друг на друга (возможны импорт и экспорт модулей)
* Могут быть использованы для реализации ленивой загрузки (загрузка по требованию) lazy loading

Каждый модуль - класс TS с декоратором @NgModule() (декоратор - подобие аннотации)

Содержит секции

* declarations - компоненты, директивы, фильтры, содержащиеся в этом модуле
* exports - то, что объявлено в этой секции, будет видно и доступно для использования в других модулях
* imports - список внешних модулей, содержимое секции exports которых используется в текущем модуле
* providers - сервисы, реализованные в этом модуле, видимые в глобальном контексте приложения
* bootstrap - главное предсталвение приложения (объявляется только в корневом модуле). С него начинается рендеринг интерфейса приложения

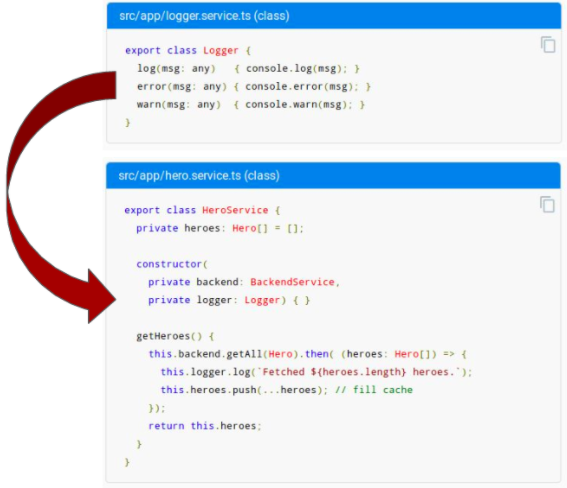
Компоненты

* У всех компонентов внутри одного модуля общий контекст компиляции. При компиляции они друг друга видят и компилятор видит все компоненты этого модуля
* Каждый компонент - отдельный класс
* Контролирует область экрана, называемую представлением (div-чик или блок)
* Ангуляр управляет жизненным циклом компонентов

Сервисы и DI

Сервисы (services) реализуют какие-либо действия, не формируя представление (логика средствами сервиса не отражается на интерфейс, потом она как-то в интерфейсе может отразиться, но не прямым путем)

* Реализуются в виде отдельых классов в соответствии с принципами ООП
* Компонент может делигировать какие-либо из своих задач сервисам
* Доступ компонентов к сервисам реализуется с помощью DI



DI - осуществляется элементом Angular Injector' ом

* Компоненты могут использовать сервисы с помощью DI
* Для того, чтобы класс можно было использовать с помощью DI, он должен содержать декоратор @Injectable()

Основные принципы реализации DI

* Приложение содержит как минимум один глобальный Injector, который занимается DI (Injector'ов может быть >1)
* Injector создает зависимости и передает их экземпляры контейнеру (container)
* Провайдер (provider) - это объект, который сообщает Injector'у, как получить или создать экземпляр зависимости
* Обычно провайдером сервиса является сам его класс
* Зависимости компонентов указываются в качестве параметров их конструкторов

1. **Angular: шаблоны страниц, жизненный цикл компонентов, подключение CSS.**

Шаблоны и представления

* Представление (view) компонента задается с помощью шаблона (template) - та самая разметка, которая будет сформирована при просмотре компонента (в браузере)
* Представления часто группируются иерархически (складываются друг в друга)
* Компонент может содержать иерархию представлений (view hierarchy), которая содержит встроенные представление (embedded views) из других компонентов

Синтаксис шаблонов

* Похож на обычный HTML
* Взаимодействие с классом компонента осуществляется с помощью ссылок на его свойства (data binding) {{hero.name}}
* Также можно использовать фильтры (pipes) и директивы (directives)

1. **Angular: клиент-серверное взаимодействие, создание, отправка и валидация данных форм.**

