# Руководство пользователя Google Guice

[**1 Зачем это нужно**](http://netvl.github.io/guice/users-guide.html#%D0%B7%D0%B0%D1%87%D0%B5%D0%BC-%D1%8D%D1%82%D0%BE-%D0%BD%D1%83%D0%B6%D0%BD%D0%BE)

Объединение компонентов программы — весьма трудоёмкая часть процесса разработки. Существует несколько подходов к связыванию классов данных, сервисов и представления в единое целое. Для иллюстрации этих подходов напишем код биллинга к сайту-магазину пиццы:

public interface BillingService {

/\*\*

\* Совершает попытку списания суммы заказа с кредитной карты. Записываются

\* как успешные, так и неуспешные транзакции.

\*

\* @return возвращает чек транзации. Если списание было успешно,

\* чек содержит подтверждение. В противном случае чек содержит

\* запись причины отказа.

\*/

Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);

}

Параллельно с настоящей реализацией напишем модульные тесты для нашего кода. Для них нам понадобится FakeCreditCardProcessor (“поддельный обработчик кредитных карт”), чтобы случайно не списать деньги с настоящей карты :)

[**1.1 Прямой вызов конструкторов**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D1%8F%D0%BC%D0%BE%D0%B9-%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D0%BE%D0%B2)

Когда мы создаём обработчик кредиток и логгер транзакций с помощью new, код выглядит так:

public class RealBillingService implements BillingService {

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

CreditCardProcessor processor = new PaypalCreditCardProcessor();

TransactionLog transactionLog = new DatabaseTransactionLog();

try {

ChargeResult result = processor.charge(creditCard, order.getAmount());

transactionLog.logChargeResult(result);

return result.wasSuccessful()

? Receipt.forSuccessfulCharge(order.getAmount())

: Receipt.forDeclinedCharge(result.getDeclineMessage());

} catch (UnreachableException e) {

transactionLog.logConnectException(e);

return Receipt.forSystemFailure(e.getMessage());

}

}

}

Такой код создаёт проблемы для тестирования и модульности. Прямая зависимость времени компиляции на реальный обработчик кредиток означает, что с карточки будут сниматься деньги, когда мы проводим тестирование! Также такой код неудобно тестировать на обработку отказа транзакции или недоступности сервиса.

[**1.2 Фабрики**](http://netvl.github.io/guice/users-guide.html#%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B8)

Фабрика отделяет клиентский код от кода реализации класса. В простой фабрике используются статические методы (геттеры/сеттеры) для получения реализации интерфейса и установки заглушки. Например, фабрика может быть реализована так:

public class CreditCardProcessorFactory {

private static CreditCardProcessor instance;

public static void setInstance(CreditCardProcessor creditCardProcessor) {

instance = creditCardProcessor;

}

public static CreditCardProcessor getInstance() {

if (instance == null) {

return new SquareCreditCardProcessor();

}

return instance;

}

}

Тогда в клиентском коде непосредственные вызовы new заменяются на запросы к фабрике:

public class RealBillingService implements BillingService {

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();

TransactionLog transactionLog = TransactionLogFactory.getInstance();

try {

ChargeResult result = processor.charge(creditCard, order.getAmount());

transactionLog.logChargeResult(result);

return result.wasSuccessful()

? Receipt.forSuccessfulCharge(order.getAmount())

: Receipt.forDeclinedCharge(result.getDeclineMessage());

} catch (UnreachableException e) {

transactionLog.logConnectException(e);

return Receipt.forSystemFailure(e.getMessage());

}

}

}

С помощью фабрик можно написать полноценный модульный тест:

public class RealBillingServiceTest extends TestCase {

private final PizzaOrder order = new PizzaOrder(100);

private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();

private final FakeCreditCardProcessor creditCardProcessor = new FakeCreditCardProcessor();

@Override public void setUp() {

TransactionLogFactory.setInstance(transactionLog);

CreditCardProcessorFactory.setInstance(creditCardProcessor);

}

@Override public void tearDown() {

TransactionLogFactory.setInstance(null);

CreditCardProcessorFactory.setInstance(null);

}

public void testSuccessfulCharge() {

RealBillingService billingService = new RealBillingService();

Receipt receipt = billingService.chargeOrder(order, creditCard);

assertTrue(receipt.hasSuccessfulCharge());

assertEquals(100, receipt.getAmountOfCharge());

assertEquals(creditCard, creditCardProcessor.getCardOfOnlyCharge());

assertEquals(100, creditCardProcessor.getAmountOfOnlyCharge());

assertTrue(transactionLog.wasSuccessLogged());

}

}

Этот код весьма неуклюж. Реализация-заглушка хранится в глобальной переменной, поэтому нам следует быть особенно осторожными при инициализации и завершении теста. Если метод tearDown выкинет исключение, то глобальная переменная по-прежнему будет указывать на тестовый экземпляр сервиса, что может вызвать проблемы в других тестах. Также нельзя запускать параллельно несколько тестов.

Однако, самой большой проблемой является то, что зависимости **скрыты в коде**. Если мы потом добавим зависимость на CreditCardFraudTracker (“трекер поддельных кредиток”), понадобится перезапускать все тесты и искать сломавшиеся. Если мы забудем проинициализировать фабрику настоящей реализацией сервиса, мы об этом не узнаем до тех пор, пока не попытаемся совершить транзакцию. С ростом кодовой базы приложения слежение за фабриками становится страшной головной болью.

Проблемы с качеством кода могут выявить QA или приёмочные тесты. Этого может быть достаточно, но мы совершенно точно можем сделать лучше.

[**1.3 Внедрение зависимостей**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9)

Как и фабрики, внедрение зависимостей — это просто паттерн проектирования. Ключевой принцип этого паттерна заключается в **отделении кода поведения от кода разрешения зависимостей**. В нашем случае это означает, что RealBillingService не должен сам создавать реализации TransactionLog и CreditCardProcessor. Вместо этого они должны передаваться как параметры конструктора:

public class RealBillingService implements BillingService {

private final CreditCardProcessor processor;

private final TransactionLog transactionLog;

public RealBillingService(CreditCardProcessor processor,

TransactionLog transactionLog) {

this.processor = processor;

this.transactionLog = transactionLog;

}

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

try {

ChargeResult result = processor.charge(creditCard, order.getAmount());

transactionLog.logChargeResult(result);

return result.wasSuccessful()

? Receipt.forSuccessfulCharge(order.getAmount())

: Receipt.forDeclinedCharge(result.getDeclineMessage());

} catch (UnreachableException e) {

transactionLog.logConnectException(e);

return Receipt.forSystemFailure(e.getMessage());

}

}

}

Теперь нам не нужны фабрики, и мы также можем упростить тест, выкинув методы setUp и tearDown:

public class RealBillingServiceTest extends TestCase {

private final PizzaOrder order = new PizzaOrder(100);

private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();

private final FakeCreditCardProcessor creditCardProcessor = new FakeCreditCardProcessor();

public void testSuccessfulCharge() {

RealBillingService billingService

= new RealBillingService(creditCardProcessor, transactionLog);

Receipt receipt = billingService.chargeOrder(order, creditCard);

assertTrue(receipt.hasSuccessfulCharge());

assertEquals(100, receipt.getAmountOfCharge());

assertEquals(creditCard, creditCardProcessor.getCardOfOnlyCharge());

assertEquals(100, creditCardProcessor.getAmountOfOnlyCharge());

assertTrue(transactionLog.wasSuccessLogged());

}

}

Теперь если мы удаляем или добавляем зависимости, компилятор нам подскажет, какие тесты нужно исправить. Зависимости становятся **явно выражены в API**.

Однако теперь бремя создания зависимостей ложится на клиента BillingService. Частично его можно снять, применив паттерн снова, т.е. классы, зависящие от BillingService, могут принимать его через конструктор. Однако, для реализации самого верхнего уровня в этой цепочке желательно воспользоваться каким-либо фреймворком, иначе зависимости придётся разрешать вручную рекурсивно каждый раз, когда нам понадобится сервис:

public static void main(String[] args) {

CreditCardProcessor processor = new PaypalCreditCardProcessor();

TransactionLog transactionLog = new DatabaseTransactionLog();

BillingService billingService

= new RealBillingService(creditCardProcessor, transactionLog);

...

}

[**1.4 Внедрение зависимостей с помощью Guice**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9-%D1%81-%D0%BF%D0%BE%D0%BC%D0%BE%D1%89%D1%8C%D1%8E-guice)

Применение паттерна внедрения зависимостей позволяет писать модульный и тестируемый код, и Guice в этом помогает. Чтобы воспользоваться им в нашей ситуации, нам сначала нужно сопоставить интерфейсы с их реализациями. Это делается в модуле Guice, который является обычным Java-классом, реализующим интерфейс Module:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

bind(TransactionLog.class).to(DatabaseTransactionLog.class);

bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);

bind(BillingService.class).to(RealBillingService.class);

}

}

Также мы добавим аннотацию @Inject к конструктору RealBillingService, чтобы Guice знал, что нужно использовать именно его. Guice, пользуясь сигнатурой такого конструктора, распознает типы параметров и создаст необходимые для них объекты автоматически.

public class RealBillingService implements BillingService {

private final CreditCardProcessor processor;

private final TransactionLog transactionLog;

@Inject

public RealBillingService(CreditCardProcessor processor,

TransactionLog transactionLog) {

this.processor = processor;

this.transactionLog = transactionLog;

}

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

try {

ChargeResult result = processor.charge(creditCard, order.getAmount());

transactionLog.logChargeResult(result);

return result.wasSuccessful()

? Receipt.forSuccessfulCharge(order.getAmount())

: Receipt.forDeclinedCharge(result.getDeclineMessage());

} catch (UnreachableException e) {

transactionLog.logConnectException(e);

return Receipt.forSystemFailure(e.getMessage());

}

}

}

Наконец, объединим всё вместе. Для получения экземпляров любого из сконфигурированных классов используется класс Injector:

public static void main(String[] args) {

Injector injector = Guice.createInjector(new BillingModule());

BillingService billingService = injector.getInstance(BillingService.class);

...

}

В следующем разделе объясняется, как это работает.

[**2 Введение**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5)

При использовании паттерна внедрения зависимостей объекты принимают свои зависимости через конструктор. Поэтому, чтобы создать объект, сначала нужно создать его зависимости. Однако, чтобы создать каждую зависимость в отдельности, нужно, в свою очередь, создать /её/ зависимости. Таким образом, на самом деле, нам нужно создать *граф зависимостей*.

Создание графов объектов вручную очень трудоёмко, чревато ошибками и усложняет тестирование. Guice может создать граф объектов за нас. Но сначала Guice нужно сконфигурировать, чтобы он создал граф именно так, как нужно.

Для иллюстрации возьмём класс RealBillingService, который принимает в качестве зависимостей два интерфейса, CreditCardProcessor и TransactionLog. Чтобы обозначить, что конструктор RealBillingService’а должен вызываться Guice’ом, мы помечаем его аннотацией @Inject:

class RealBillingService implements BillingService {

private final CreditCardProcessor processor;

private final TransactionLog transactionLog;

@Inject

RealBillingService(CreditCardProcessor processor, TransactionLog transactionLog) {

this.processor = processor;

this.transactionLog = transactionLog;

}

@Override

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

...

}

}

Мы хотим создать RealBillingService с вполне конкретными зависимостями: PaypalCreditCardProcessor и DatabaseTransactionLog. Guice использует **привязки** (**биндинги**), чтобы установить соответствие между типами (интерфейсами) и их конкретными реализациями. **Модуль** — это набор привязок, выраженных с помощью вызовов методов, напоминающих простой английский:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

/\*

\* Здесь мы указываем Guice'у, что где бы он не увидел зависимость на интерфейс TransactionLog,

\* он должен её разрешить с использованием класса DatabaseTransactionLog.

\*/

bind(TransactionLog.class).to(DatabaseTransactionLog.class);

/\*

\* Аналогично, эта привязка указывает, что если интерфейс CreditCardProcessor используется

\* где-либо в качестве зависимости, Guice должен будет использовать класс PaypalCreditCardProcessor.

\*/

bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);

}

}

Модули — это строительный материал для **инжектора** (**injector**), класса, который в Guice занимается созданием графов объектов. Сначала мы создаём инжектор с использованием модуля, а потом мы можем его использовать для получения экземпляра RealBillingService:

public static void main(String[] args) {

/\*

\* Guice.createInjector() берёт набор модулей и возвращает готовый инстанс инжектора.

\* Большинство приложений вызывают этот метод единственный раз, в коде main-метода.

\*/

Injector injector = Guice.createInjector(new BillingModule());

/\*

\* Теперь, раз у нас есть инжектор, мы можем создавать объекты.

\*/

RealBillingService billingService = injector.getInstance(RealBillingService.class);

...

}

Создав здесь billingService, мы на самом деле создали маленький граф объектов. В этом графе содержатся сам биллинговый сервис, а также обработчик кредиток и лог транзакций, от которых сервис зависит.

[**3 Привязки**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

Задача инжектора — собрать граф объектов. Мы говорим ему, что хотим получить экземпляр заданного типа, и он автоматически выясняет, что именно нужно создать, разрешает зависимости и связывает всё вместе. Чтобы указать, каким конкретно образом следует разрешать зависимости, инжектор нужно сконфигурировать с помощью привязок.

Привязки создаются с помощью модулей. Для этого проще всего унаследовать абстрактный класс AbstractModule и переопределить его метод configure(). В теле этого метода нужно вызывать метод bind() чтобы описать каждую привязку. Эти методы проверяются компилятором на наличие ошибок типов, поэтому компилятор подскажет нам, если мы ошиблись в типах. Как только мы описали модуль, его можно передать в метод Guice.createInjector(), чтобы создать инжектор.

Наиболее используемыми являются следующие виды привязок: *компоновочные* (*linked*), *экземплярные* (*instance*), *@Provides-методы*, *привязки провайдеров* (*providers*), *привязки конструкторов* (*constructor bindings*) и *бесцелевые* (*untargetted*) *привязки*.

Помимо этого, Guice поддерживает некоторое количество *встроенных привязок*. Также, если инжектор обнаружил зависимость, привязка для которой не описан в его конфигурации, он попытается создать *неявную* (*just-in-time*) *привязку*. Ещё Guice умеет внедрять зависимости на провайдеров к другим привязкам.

[**3.1 Компоновочные привязки**](http://netvl.github.io/guice/users-guide.html#%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D0%BE%D1%87%D0%BD%D1%8B%D0%B5-%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

**Компоновочные** (или **обычные**) привязки связывают тип с его реализацией. В следующем примере интерфейс TransactionLog связывется со своей реализацией DatabaseTransactionLog:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

bind(TransactionLog.class).to(DatabaseTransactionLog.class);

}

}

Теперь, когда мы вызываем Injector.getInstance(TransactionLog.class), либо если инжектор сам обнаружит в графе объектов зависимость от TransactionLog, то он автоматически будет использовать DatabaseTransactionLog. В компоновочных привязках связь может идти от типа к любому из подтипов, например, от интерфейса к его реализации или от класса к его наследнику.

Компоновочные привязки могут образовывать цепочки:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

bind(TransactionLog.class).to(DatabaseTransactionLog.class);

bind(DatabaseTransactionLog.class).to(MySqlDatabaseTransactionLog.class);

}

}

В таком случае, когда нам или инжектору понадобится реализация интерфейса TransactionLog, будет использован класс MySqlDatabaseTransactionLog.

[**3.2 Привязочные аннотации**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BE%D1%87%D0%BD%D1%8B%D0%B5-%D0%B0%D0%BD%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D0%B8)

Иногда требуется создать несколько привязок одного и того же типа. Например, одновременно в программе может потребоваться использовать обработчик кредитных кард PayPal и обработчик Google Checkout. Для этого используются **привязочные аннотации**. Аннотация и тип вместе определяют привязку единственным образом. Такая пара называется **ключом**.

Определение привязочной аннотации потребует нескольких строк кода и импортов. Поместим следующий код в отдельный java-файл:

package example.pizza;

import com.google.inject.BindingAnnotation;

import java.lang.annotation.Target;

import java.lang.annotation.Retention;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

import static java.lang.annotation.ElementType.PARAMETER;

import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.ElementType.METHOD;

@BindingAnnotation

@Target({FIELD, PARAMETER, METHOD})

@Retention(RUNTIME)

public @interface PayPal {

}

Знать, что означают все эти дополнительные мета-аннотации, необязательно, но если интересно, то вот описание: - @BindingAnnotation указывает Guice, что это привязочная аннотация; если один и тот же идентификатор помечен двумя и более привязочными аннотациями, то Guice выкинет ошибку; - @Target({FIELD, PARAMETER, METHOD}) указывает, где можно применять аннотацию; это не даст случайно применить @PayPal где-то, где это не имеет смысла; - @Retention(RUNTIME) делает аннотацию доступной для анализа во время выполнения программы.

Чтобы объявить зависимость на аннотированную привязку, достаточно просто применить аннотацию на внедряемый параметр:

public class RealBillingService implements BillingService {

@Inject

public RealBillingService(@PayPal CreditCardProcessor processor, TransactionLog transactionLog) {

...

}

}

Чтобы объявить саму аннотированную привязку, нужно воспользоваться необязательной конструкцией annotatedWith в объявлении bind():

bind(CreditCardProcessor.class)

.annotatedWith(PayPal.class)

.to(PayPalCreditCardProcessor.class);

[**3.2.1 @Named**](http://netvl.github.io/guice/users-guide.html#named)

По умолчанию в Guice есть аннотация @Named, которая принимает строку в качестве параметра:

public class RealBillingService implements BillingService {

@Inject

public RealBillingService(@Named("Checkout") CreditCardProcessor processor, TransactionLog transactionLog) {

...

}

}

Чтобы воспользоваться такой привязкой, нужно воспользоваться методом =Names.named()=, передав результат его вызова в annotatedWith:

bind(CreditCardProcessor.class)

.annotatedWith(Names.named("Checkout"))

.to(CheckoutCreditCardProcessor.class);

Поскольку компилятор не проверяет строки, лучше использовать @Named как можно реже.

[**3.2.2 Привязочные аннотации с атрибутами**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BE%D1%87%D0%BD%D1%8B%D0%B5-%D0%B0%D0%BD%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D0%B8-%D1%81-%D0%B0%D1%82%D1%80%D0%B8%D0%B1%D1%83%D1%82%D0%B0%D0%BC%D0%B8)

Guice поддерживает создание и использование привязочных аннотаций, с которыми связаны конкретные значения (как у @Named). В тех редких случаях, когда это может понадобиться, нужно сделать следующее: 1. создать интерфейс аннотации; 2. создать класс, “реализующий” интерфейс аннотации; при определении методов hashCode() и equals() следует пользоваться [документацией](http://java.sun.com/javase/6/docs/api/java/lang/annotation/Annotation.html); 3. передать экземпляр этого класса в метод annotatedWith() при объявлении привязки.

[**3.3 Экземплярные привязки**](http://netvl.github.io/guice/users-guide.html#%D1%8D%D0%BA%D0%B7%D0%B5%D0%BC%D0%BF%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5-%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

Тип можно привязать к конкретному объекту, принадлежащему этому типу. Это полезно только в том случае, когда требуется привязывать объекты без собственных зависимостей, такие, как объекты-значения.

bind(String.class)

.annotatedWith(Names.named("JDBC URL"))

.toInstance("jdbc:mysql://localhost/pizza");

bind(Integer.class)

.annotatedWith(Names.named("login timeout seconds"))

.toInstance(10);

Избегайте использования экземплярных привязок с объектами, которые относительно сложно создавать, потому что это может замедлить загрузку вашего приложения. Вместо этого можно использовать @Provides-методы.

[**3.4 @Provides-методы**](http://netvl.github.io/guice/users-guide.html#provides-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D1%8B)

Когда нужно сконструировать объект-зависимость нестандартным способом, можно воспользоваться @Provides-методом. Этот метод должен быть объявлен в модуле и должен быть аннотирован как @Provides. Такой метод обявляет привязку аналогично вызову метода bind(). Возвращаемый тип метода становится типом привязки; когда инжектору понадобится зависимость такого типа, будет вызван этот метод.

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

...

}

@Provides

TransactionLog provideTransactionLog() {

DatabaseTransactionLog transactionLog = new DatabaseTransactionLog();

transactionLog.setJdbcUrl("jdbc:mysql://localhost/pizza");

transactionLog.setThreadPoolSize(30);

return transactionLog;

}

}

Если помимо @Provides на методе присутствует ещё некоторая привязочная аннотация, например, @Named("Checkout") или @PayPal, Guice создаст аннотированную привязку.

Зависимости для создания объекта можно передать через параметры метода. Перед его вызовом инжектор их разрешит:

@Provides @PayPal

CreditCardProcessor providePayPalCreditCardProcessor(@Named("PayPal API key") String apiKey) {

PayPalCreditCardProcessor processor = new PayPalCreditCardProcessor();

processor.setApiKey(apiKey);

return processor;

}

[**3.4.1 Выбрасывание исключений**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D1%8B%D0%B1%D1%80%D0%B0%D1%81%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9)

Guice не позволяет выбрасывать исключения из @Provides-методов. Все исключения, выброшенные этими методами, будут оборачиваться в ProvisionException. Выбрасывание исключений из @Provides-методов, как checked-, так и unchecked- — плохая практика. Если всё же для чего-то это понадобится, то можно воспользоваться расширением *ThrowingProviders* и аннотацией, которую оно предоставляет, @CheckedProvides.

[**3.5 Привязки провайдеров**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8-%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D0%BE%D0%B2)

Как только @Provides-методы становятся достаточно сложными, появляется смысл вынести их в отдельные классы. Такие классы-провайдеры должны реализовывать очень простой интерфейс Provider: java public interface Provider<T> { T get(); }

Реализации этого интерфейса могут иметь свои зависимости, которые можно получить через помеченный аннотацией @Inject конструктор.

public class DatabaseTransactionLogProvider implements Provider<TransactionLog> {

private final Connection connection;

@Inject

public DatabaseTransactionLogProvider(Connection connection) {

this.connection = connection;

}

public TransactionLog get() {

DatabaseTransactionLog transactionLog = new DatabaseTransactionLog();

transactionLog.setConnection(connection);

return transactionLog;

}

}

Теперь провайдер можно привязать с помощью конструкции toProvider:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

bind(TransactionLog.class)

.toProvider(DatabaseTransactionLogProvider.class);

}

}

Если провайдеры достаточно сложны, на них тоже следует писать модульные тесты.

[**3.5.1 Выбрасывание исключений**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D1%8B%D0%B1%D1%80%D0%B0%D1%81%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9-1)

Guice не позволяет выбрасывать исключения из провайдеров. Интерфейс Provider запрещает использование checked-исключений. Unchecked-исключения будут обёрнуты в ProvisionException или в CreationException, что может помешать созданию инжектора. Если по какой-то причине вам всё же понадобится выкидывать исключения из провайдеров, можно воспользоваться расширением *ThrowingProviders*.

[**3.6 Бесцелевые привязки**](http://netvl.github.io/guice/users-guide.html#%D0%B1%D0%B5%D1%81%D1%86%D0%B5%D0%BB%D0%B5%D0%B2%D1%8B%D0%B5-%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

Можно создавать привязки без указания цели. Наиболее полезно это при использовании классов, помеченных аннотациями @ImplementedBy или @ProvidedBy. Бесцелевая привязка сообщает инжектору, что ему необходимо заранее подготовить зависимости для указанного класса. Такие привязки объявляются без конструкции to():

bind(MyConcreteClass.class);

bind(AnotherConcreteClass.class).in(Singleton.class);

Если используются привязочные аннотации, то цель привязки нужно указывать в любом случае, даже если класс привязывается сам к себе:

bind(MyConcreteClass.class).annotatedWith(Names.named("foo")).to(MyConcreteClass.class);

bind(AnotherConcreteClass.class).annotatedWith(Names.named("foo")).to(AnotherConcreteClass.class).in(Singleton.class);

[**3.7 Привязки конструкторов**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D0%BE%D0%B2)

Иногда бывает необходимо привязать тип к некоторому конкретному конструктору. Например, может быть так, что аннотацией @Inject нельзя пометить нужный конструктор, потому что класс принадлежит сторонней библиотеке, либо из-за того, что для процесса внедрения зависимостей используется несколько конструкторов. @Provides-методы решают эти проблемы наилучшим образом: вызывая конструктор явно, не требуется прибегать к отражению и связанных с ним проблем. Но у этого подхода есть ограничения, в частности, вручную созданные объекты не могут участвовать в АОП.

В качестве компенcации Guice предоставляет привязки toConstructor(). Для их использования нужно получить конструктор с помощью отражения и обработать исключение, выбрасываемое, когда такого конструктора нет:

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

try {

bind(TransactionLog.class).toConstructor(

DatabaseTransactionLog.class.getConstructor(DatabaseConnection.class));

} catch (NoSuchMethodException e) {

addError(e);

}

}

}

Из примера видно, что у класса DatabaseTransactionLog должен быть конструктор, который принимает единственный параметр типа DatabaseConnection. Аннотация @Inject на этом конструкторе необязательна. Для разрешения зависимости Guice вызовет указанный конструктор.

Каждая привязка конструктора обладает собственной областью видимости. Если создать несколько синглтонных привязок к одному и тому же конструктору, каждая будет давать собственный экземпляр класса.

[**3.8 Встроенные привязки**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D1%81%D1%82%D1%80%D0%BE%D0%B5%D0%BD%D0%BD%D1%8B%D0%B5-%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

Кроме явных и неявных (just-in-time) привязок инжектор Guice автоматически объявляет некоторые стандартные привязки. Только инжектор может объявлять такие привязки; попытка создать их вручную приведёт к ошибке.

[**3.8.1 Логгеры**](http://netvl.github.io/guice/users-guide.html#%D0%BB%D0%BE%D0%B3%D0%B3%D0%B5%D1%80%D1%8B)

Guice предлагает встроенную привязку к java.util.logging.Logger для сокращения шаблонного кода. Эта привязка автоматически устанавливает в качестве имени логгера имя класса, в который внедряется логгер.

@Singleton

public class ConsoleTransactionLog implements TransactionLog {

private final Logger logger;

@Inject

public ConsoleTransactionLog(Logger logger) {

this.logger = logger;

}

public void logConnectException(UnreachableException e) {

/\* the message is logged to the "ConsoleTransacitonLog" logger \*/

logger.warning("Connect exception failed, " + e.getMessage());

}

}

[**3.8.2 Инжектор**](http://netvl.github.io/guice/users-guide.html#%D0%B8%D0%BD%D0%B6%D0%B5%D0%BA%D1%82%D0%BE%D1%80)

Бывает так (например, в случае написания фреймворков), что используемые типы неизвестны до времени выполнения. В этих редких ситуациях можно внедрять сам инжектор. Зависимости кода, который использует инжектор как зависимость, становятся неявными, поэтому такой подход следует использовать как можно реже.

[**3.8.3 Провайдеры**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D1%8B)

Для любого типа, который известен Guice, можно внедрять провайдер этого типа. Детали такого способа описаны далее.

[**3.8.4 TypeLiteral’ы**](http://netvl.github.io/guice/users-guide.html#typeliteral%D1%8B)

Guice обладает полной информацией о типах всех внедряемых классов. Если требуется внедрить параметризованный тип, то вместе с ним можно внедрить экземпляр класса TypeLiteral<T>, чтобы получить информацию о типе во время выполнения.

[**3.8.5 Стадии**](http://netvl.github.io/guice/users-guide.html#%D1%81%D1%82%D0%B0%D0%B4%D0%B8%D0%B8)

Guice поддерживает стадии, которые нужны для того, чтобы можно было различать зависимости времени разработки и продакшена.

[**3.8.6 MembersInjector’ы**](http://netvl.github.io/guice/users-guide.html#membersinjector%D1%8B)

При использовании привязок к провайдерам или при написании расширений иногда требуется внедрять зависимости в объект, созданный вручную. Для этого можно объявить зависимость на класс MembersInjector<T>, где T — тип объекта, а затем вызывать метод membersInjector.injectMembers(myNewObject).

[**3.9 Неявные (just-in-time) привязки**](http://netvl.github.io/guice/users-guide.html#%D0%BD%D0%B5%D1%8F%D0%B2%D0%BD%D1%8B%D0%B5-just-in-time-%D0%BF%D1%80%D0%B8%D0%B2%D1%8F%D0%B7%D0%BA%D0%B8)

Для того, чтобы создать объект некоторого типа, инжектору нужно описание, как имено это сделать, то есть привязка. Привязки, заданные в модулях, называются *явными*, и инжектор по возможности использует их. Если же требуется создать объект типа, который не привязан явно, инжектор попробует создать *неявную* (*just-in-time*) *привязку*. Другими словами, инжектор попытается создать нужный объект, применив некоторую эвристику, позволяющую отыскать подходящую реализацию типа.

[**3.9.1 Инжекционные конструкторы**](http://netvl.github.io/guice/users-guide.html#%D0%B8%D0%BD%D0%B6%D0%B5%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D1%8B)

Guice может создавать привязки для конкретных типов, используя их *инжекционный* конструктор. Это может быть либо публичный конструктор без параметров, либо конструктор, помеченный аннотацией @Inject:

public class PayPalCreditCardProcessor implements CreditCardProcessor {

private final String apiKey;

@Inject

public PayPalCreditCardProcessor(@Named("PayPal API key") String apiKey) {

this.apiKey = apiKey;

}

}

Guice не будет создавать нестатические вложенные классы, поскольку у них есть неявная ссылка на экземпляр внешнего класса, которую нельзя внедрить.

[**3.9.2 @ImplementedBy**](http://netvl.github.io/guice/users-guide.html#implementedby)

Типы можно помечать аннотациями, чтобы указать инжектору, какую реализацию типа по умолчанию нужно использовать. Аннотация @ImplementedBy, указывая подтип, который нужно использовать для внедрения данного типа, действует как **компоновочная привязка**:

@ImplementedBy(PayPalCreditCardProcessor.class)

public interface CreditCardProcessor {

ChargeResult charge(String amount, CreditCard creditCard)

throws UnreachableException;

}

Аннотация в этом примере эквивалентна следующей конструкции bind():

bind(CreditCardProcessor.class).to(PayPalCreditCardProcessor.class);

Если тип как участвует в bind() (в качестве первого аргумента), так и помечен аннотацией @ImplementedBy, то будет использована привязка через bind(). Аннотация же обозначает некоторую реализацию по умолчанию, которую можно переопределить с помощью явной привязки. @ImplementedBy следует использовать с осторожностью, потому что она создаёт зависимость времени компиляции от интерфейса к его реализации.

[**3.9.3 @ProvidedBy**](http://netvl.github.io/guice/users-guide.html#providedby)

Аннотация @ProvidedBy указывает инжектору, что следует использовать указанный класс провайдера для создания объектов типа:

@ProvidedBy(DatabaseTransactionLogProvider.class)

public interface TransactionLog {

void logConnectException(UnreachableException e);

void logChargeResult(ChargeResult result);

}

Эта аннотация эквивалентна следующей привязке к провайдеру:

bind(TransactionLog.class)

.toProvider(DatabaseTransactionLogProvider.class);

Как и в случае с @ImplementedBy, если тип одновременно аннотирован @ProvidedBy и привязан с помощью конструкции bind(), выигрывает последняя.

[**4 Области видимости**](http://netvl.github.io/guice/users-guide.html#%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8)

По умолчанию инжектор Guice каждый раз при внедрении создаёт новый объект. Это поведение настраивается с помощью *областей видимости* (*scopes*). Области видимости позволяют использовать одни и те же объекты много раз: за всё время работы приложения (@Singleton), за время сессии (@SessionScoped) или запроса (@RequestScoped). В комплект Guice входит расширение, позволяющее создавать сервлеты и определяющее области видимости для веб-приложений. Также возможно создавать свои области видимости.

[**4.1 Применение областей видимости**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D0%B9-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8)

Для определения областей в Guice используются аннотации. Область видимости для типа можно указать, пометив аннотацией класс-реализацию. В таком случае аннотация служит дополнительной документацией. Например, из наличия аннотации @Singleton следует, что класс должен быть потокобезопасным.

@Singleton

public class InMemoryTransactionLog implements TransactionLog {

/\* всё, что здесь объявлено, должно быть потокобезопасным! \*/

}

Также области видимости можно настраивать в конструкции bind():

bind(TransactionLog.class).to(InMemoryTransactionLog.class).in(Singleton.class);

А также аннотацией на @Provides-методах:

@Provides @Singleton

TransactionLog provideTransactionLog() {

...

}

Если области видимости, указанные непосредственно на классе и с помощью bind(), конфликтуют, то будет выбрана та, что указана с помощью bind(). Если тип аннотирован некоторой областью, а нужно от этого избавиться, то достаточно привязать его с помощью bind() с областью Scopes.NO\_SCOPE.

В компоновочных привязках области видимости применяются к типу, который привязывается к реализации, а не наоборот. Например, пусть у нас есть класс Applebees, который реализует интерфейсы Bar и Grill. Следующие привязки позволят создать **два** экземпляра этого класса, по одному для каждого интерфейса, несмотря на то, что они находятся в области @Singleton:

bind(Bar.class).to(Applebees.class).in(Singleton.class);

bind(Grill.class).to(Applebees.class).in(Singleton.class);

Так происходит именно потому, что области видимости применяются к типу, к которому делается привязка (Bar, Grill), но не к тому, который привязывается сам (Applebees). Чтобы создать только один экземпляр Applebees даже в этом случае, следует либо пометить Applebees аннотацией @Singleton, либо объявить привязку

bind(Applebees.class).in(Singleton.class);

Если объявлена такая привязка, то конструкции in(Singleton.class) в тех привязках, что указаны выше, необязательны.

Конструкция in() принимает либо класс аннотации области видимости (например, RequestScoped.class), либо экземпляр класса, “реализующего” интерфейс соответствующей аннотации, например, ServletScope.REQUEST:

bind(UserPreferences.class)

.toProvider(UserPreferencesProvider.class)

.in(ServletScopes.REQUEST);

Использование непосредственно классов аннотаций предпочтительно, потому что это позволяет использовать тот же модуль повторно в других типах приложений. Например, объект, помеченный @RequestScope, может находиться в области видимости HTTP-запроса в веб-приложении или в области видимости RPC-вызова в сервере API.

[**4.2 Энергичные синглтоны**](http://netvl.github.io/guice/users-guide.html#%D1%8D%D0%BD%D0%B5%D1%80%D0%B3%D0%B8%D1%87%D0%BD%D1%8B%D0%B5-%D1%81%D0%B8%D0%BD%D0%B3%D0%BB%D1%82%D0%BE%D0%BD%D1%8B)

Guice предоставляет специальный синтаксис для определения синглтонов, которые должны создаваться сразу при конфигурации:

bind(TransactionLog.class).to(InMemoryTransactionLog.class).asEagerSingleton();

В случае использования энергичных синглтонов проблемы с их инициализацией всплывают раньше, что с точки зрения клиента удобнее и логичнее. С ленивыми синглтонами, однако, цикл редактирования-компиляции-запуска при разработке проходит быстрее. Для выбора подходящей стратегии можно использовать перечисление Stage:

+———————–+——————+——————-+ | | Stage.PRODUCTION | Stage.DEVELOPMENT | |=======================+==================+===================| | .asEagerSingleton() | энергичная | энергичная | | .in(Singleton.class) | энергичная | ленивая | | .in(Scopes.SINGLETON) | энергичная | ленивая | | @Singleton | энергичная\* | ленивая | +———————–+——————+——————-+ \* Guice будет создавать синглтоны энергично только для тех типов, о которых он знает, то есть, для типов, указанных в модулях, переданных инжектору, а также для их транзитивных зависимостей.

[**4.3 Выбор области видимости**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D1%8B%D0%B1%D0%BE%D1%80-%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8)

Если рассматриваемый объект **имеет состоянием**, то выбор области видимости очевиден: если состояние должно распространяться на всё приложение, то следует использовать @Singleton; если на один запрос, то @RequestScoped, и т.д. Если объект **не имеет состояния**, а также если его создание **дёшево**, то задание области видимости необязательно. Если не указывать области видимости для привязки, Guice будет создавать новые экземпляры объектов каждый раз, когда они потребуются.

Использование синглтонов при разработке Java-приложений довольно популярно, но синглтоны не так уж и нужны, особенно в контексте использования внедрения зависимостей. Хоть синглтоны и позволяют избежать накладных расходов на создание объекта и последующее его удаление сборщиком мусора, использование экземпляра синглтона требует синхронизации. Наиболее полезны синглтоны в следующих случаях: - объекты с состоянием, такие, как конфигурация или счётчики; - объекты, которые дорого создавать; - объекты, которые используют внешние ресурсы, например, пул соединений к базе данных.

[**4.4 Области видимости и многопоточность**](http://netvl.github.io/guice/users-guide.html#%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8-%D0%B8-%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C)

Классы, помеченные аннотациями @Singleton или @SessionScoped, **должны быть потокобезопасны**. Все их зависимости также должны быть потокобезопасны. Для достижения этого следует ограничить изменяемое состояние до минимума.

@RequestScoped-объекты не обязательно должны быть синхронизированы. Обычно наличие зависимости от @Singleton- или @SessionScoped-объекты на @RequestScoped-объект — это ошибка проектирования. Если требуется получить объект из более узкой области видимости, следует внедрить провайдер нужного типа (о внедрении провайдеров ниже).

[**5 Внедрение зависимостей**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9-1)

Паттерн внедрения зависимостей отделяет код поведения от кода разрешения зависимостей. Вместо того, чтобы создавать зависимости явно или с помощью фабрик, класс должен принимать их извне. Этот процесс установки зависимостей в объект называется *внедрением*.

[**5.1 Способы внедрения зависимостей**](http://netvl.github.io/guice/users-guide.html#%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D1%8B-%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D1%8F-%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9)

[**5.1.1 Внедрение через конструктор**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D1%87%D0%B5%D1%80%D0%B5%D0%B7-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80)

Внедрение через конструктор совмещает процессы создания объекта и внедрения его зависимостей. Чтобы воспользоваться этим типом внедрения, следует пометить конструктор аннотацией @Inject. Этот конструктор должен принимать в качестве параметров все зависимости класса. Обычно такие конструкторы сразу сохраняют все свои параметры в final-полях объекта:

public class RealBillingService implements BillingService {

private final CreditCardProcessor processorProvider;

private final TransactionLog transactionLogProvider;

@Inject

public RealBillingService(CreditCardProcessor processorProvider,

TransactionLog transactionLogProvider) {

this.processorProvider = processorProvider;

this.transactionLogProvider = transactionLogProvider;

}

}

Если у класса нет конструктора, помеченного @Inject, Guice воспользуется публичным конструктором без параметров, если он есть. Однако, лучше использовать именно аннотированный конструктор, потому что он служит дополнительной документацией, предоставляя явный список зависимостей класса.

Внедрение через конструктор очень хорошо работает при модульном тестировании. Если класс принимает все свои зависимости через единственный конструктор, нельзя случайно забыть передать ему какую-то зависимость. При появлении новой зависимости весь клиентский код перестаёт компилироваться, что очень удобно: достаточно исправить ошибки компиляции, и можно быть уверенным, что всё работает правильно.

[**5.1.2 Внедрение через методы**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D1%87%D0%B5%D1%80%D0%B5%D0%B7-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D1%8B)

Для внедрения зависимостей Guice также может пользоваться методами, помеченными @Inject. Зависимости в таком случае передаются через параметры метода; инжектор их автоматически разрешит перед тем, как вызвать метод. Такие методы могут иметь произвольное число параметров, а имя их не влияет на процесс внедрения.

public class PayPalCreditCardProcessor implements CreditCardProcessor {

private static final String DEFAULT\_API\_KEY = "development-use-only";

private String apiKey = DEFAULT\_API\_KEY;

@Inject

public void setApiKey(@Named("PayPal API key") String apiKey) {

this.apiKey = apiKey;

}

}

[**5.1.3 Внедрение в поля**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-%D0%BF%D0%BE%D0%BB%D1%8F)

Guice может внедрять зависимости непосредственно в поля, помеченные аннотацией @Inject. Это наиболее краткая и выразительная форма объявления зависимостей, но она также и наиболее подвержена проблемам при тестировании.

public class DatabaseTransactionLogProvider implements Provider<TransactionLog> {

@Inject Connection connection;

public TransactionLog get() {

return new DatabaseTransactionLog(connection);

}

}

Следует избегать использования внедрения в final-поля, потому что семантика изменения final-полей плохо определена и может быть различна на разных JVM.

[**5.1.4 Необязательные внедрения**](http://netvl.github.io/guice/users-guide.html#%D0%BD%D0%B5%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5-%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D1%8F)

Иногда бывает удобно пользоваться конкретной зависимостью, если она есть, а если её нет, то использовать некоторый объект по умолчанию. Внедрения в методы или в поля могут быть *необязательными*, то есть Guice будет их игнорировать, если соответствующие зависимости недоступны. Чтобы воспользоваться необязательным внедрениям, нужно пометить метод или поле аннотацией @Inject(optional=true):

public class PayPalCreditCardProcessor implements CreditCardProcessor {

private static final String SANDBOX\_API\_KEY = "development-use-only";

private String apiKey = SANDBOX\_API\_KEY;

@Inject(optional=true)

public void setApiKey(@Named("PayPal API key") String apiKey) {

this.apiKey = apiKey;

}

}

Использование одновременно необязательных внедрений и неявных привязок может привести к неожиданному результату. Например, в следующем примере в поле всегда внедряется объект Date, даже если привязка к нему не объявлена явно. Так происходит из-за того, что у Date есть публичный конструктор без параметров, подходящий для неявных привязок.

@Inject(optional=true) Date launchDate;

[**5.1.5 Внедрение по запросу**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BF%D0%BE-%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D1%83)

Внедрения в поля и методы можно совершать для уже существующих объектов, даже не созданных Guice. Для этого используется метод Injector.injectMembers():

public static void main(String[] args) {

Injector injector = Guice.createInjector(...);

CreditCardProcessor creditCardProcessor = new PayPalCreditCardProcessor();

injector.injectMembers(creditCardProcessor);

}

[**5.1.6 Статические внедрения**](http://netvl.github.io/guice/users-guide.html#%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5-%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D1%8F)

Guice даёт возможность постепенно мигрировать со статических фабрик на внедрение зависимостей. В этом помогают *статические внедрения*. Они позволяют объектам участвовать в процессе внедрения зависимостей *лишь частично*, предоставляя им доступ к внедряемым значениям, но не применяя внедрение к ним самим. Для этого нужно воспользоваться методом requestStaticInjection() в каком-либо модуле, передав в него список классов, которые требуется проинициализировать на этапе создания инжектора:

@Override public void configure() {

requestStaticInjection(ProcessorFactory.class);

...

}

Guice внедрит зависимости в статические члены класса, помеченные @Inject:

class ProcessorFactory {

@Inject static Provider<Processor> processorProvider;

/\*\*

\* @deprecated prefer to inject your processor instead.

\*/

@Deprecated

public static Processor getInstance() {

return processorProvider.get();

}

}

Статические члены не будут внедряться во время внедрения экземпляров объектов. Этот API не предназначен для постоянного использования, потому что он приводит к тем же проблемам, что и статические фабрики: код с его использованием трудно тестировать, зависимости становятся непрозрачными, а также используется глобальное состояние.

[**5.1.7 Автоматическое внедрение**](http://netvl.github.io/guice/users-guide.html#%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5-%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5)

Guice автоматически внедряет следующие объекты: - объекты, переданные в метод toInstance() в конструкции bind(); - экземпляры провайдеров, переданные в метод toProvider() в конструкции bind().

Эти объекты будут внедряться во время создания самого инжектора. Если они нужны для создания других стартовых зависимостей, Guice их внедрит правильным образом.

[**5.2 Внедрение провайдеров**](http://netvl.github.io/guice/users-guide.html#%D0%B2%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D0%BE%D0%B2)

При обычном внедрении зависимостей каждый тип получает в точности один экземпляр каждой из своих зависимостей. Например, RealBillingService получит один CreditCardProcessor и один TransactionLog. Когда требуется большая гибкость, Guice позволяет внедрять *провайдеры*. Провайдеры создают объект, когда на них вызывается метод get():

public interface Provider<T> {

T get();

}

Тип провайдера *параметризован*; это нужно для того, чтобы отличить, например, Provider<TransactionLog> от Provider<CreditCardProcessor>. В любом месте, где используется внедрение объекта, можно вместо этого воспользоваться внедрением провайдера:

public class RealBillingService implements BillingService {

private final Provider<CreditCardProcessor> processorProvider;

private final Provider<TransactionLog> transactionLogProvider;

@Inject

public RealBillingService(Provider<CreditCardProcessor> processorProvider,

Provider<TransactionLog> transactionLogProvider) {

this.processorProvider = processorProvider;

this.transactionLogProvider = transactionLogProvider;

}

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

CreditCardProcessor processor = processorProvider.get();

TransactionLog transactionLog = transactionLogProvider.get();

/\* use the processor and transaction log here \*/

}

}

Это возможно, так как для каждой привязки, как аннотированной, так и нет, создаётся неявная привязка для провайдера.

[**5.2.1 Провайдеры для создания нескольких объектов**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D1%8B-%D0%B4%D0%BB%D1%8F-%D1%81%D0%BE%D0%B7%D0%B4%D0%B0%D0%BD%D0%B8%D1%8F-%D0%BD%D0%B5%D1%81%D0%BA%D0%BE%D0%BB%D1%8C%D0%BA%D0%B8%D1%85-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BE%D0%B2)

Провайдеры можно использовать, когда требуется создать несколько объектов одного типа. Пусть приложение сохраняет запись с итогом и деталями операции, когда оплата за пиццу не проходит. С помощью провайдеров можно создавать новые записи тогда, когда это нужно:

public class LogFileTransactionLog implements TransactionLog {

private final Provider<LogFileEntry> logFileProvider;

@Inject

public LogFileTransactionLog(Provider<LogFileEntry> logFileProvider) {

this.logFileProvider = logFileProvider;

}

public void logChargeResult(ChargeResult result) {

LogFileEntry summaryEntry = logFileProvider.get();

summaryEntry.setText("Charge " + (result.wasSuccessful() ? "success" : "failure"));

summaryEntry.save();

if (!result.wasSuccessful()) {

LogFileEntry detailEntry = logFileProvider.get();

detailEntry.setText("Failure result: " + result);

detailEntry.save();

}

}

}

[**5.2.2 Провайдеры для ленивой инициализации**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D1%8B-%D0%B4%D0%BB%D1%8F-%D0%BB%D0%B5%D0%BD%D0%B8%D0%B2%D0%BE%D0%B9-%D0%B8%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8)

Если у какого-то класса есть зависимость, которую достаточно дорого создавать, можно воспользоваться провайдером, чтобы отложить этот процесс до того момента, когда это понадобится. Это особенно полезно, если такая зависимость используется не всегда.

public class DatabaseTransactionLog implements TransactionLog {

private final Provider<Connection> connectionProvider;

@Inject

public DatabaseTransactionLog(Provider<Connection> connectionProvider) {

this.connectionProvider = connectionProvider;

}

public void logChargeResult(ChargeResult result) {

/\* only write failed charges to the database \*/

if (!result.wasSuccessful()) {

Connection connection = connectionProvider.get();

}

}

}

[**5.2.3 Провайдеры для доступа к более узким областям видимости**](http://netvl.github.io/guice/users-guide.html#%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B9%D0%B4%D0%B5%D1%80%D1%8B-%D0%B4%D0%BB%D1%8F-%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%B0-%D0%BA-%D0%B1%D0%BE%D0%BB%D0%B5%D0%B5-%D1%83%D0%B7%D0%BA%D0%B8%D0%BC-%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%8F%D0%BC-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8)

Зависимость класса из более широкой области видимости на класс из более узкой ошибочна. Если, например, лог транзакций — это синглтон, которому для работы требуется знать текущего пользователя, и если внедрить объект, представляющий пользователя, непосредственно в лог транзакций, то программа не будет работать, потому что для каждого запроса пользователь может быть разный, а в лог внедрение делается только один раз. Поскольку провайдеры могут создавать значения только по необходимости, с их помощью можно безопасно получать доступ к более узкой области видимости:

@Singleton

public class ConsoleTransactionLog implements TransactionLog {

private final AtomicInteger failureCount = new AtomicInteger();

private final Provider<User> userProvider;

@Inject

public ConsoleTransactionLog(Provider<User> userProvider) {

this.userProvider = userProvider;

}

public void logConnectException(UnreachableException e) {

failureCount.incrementAndGet();

User user = userProvider.get();

System.out.println("Connection failed for " + user + ": " + e.getMessage());

System.out.println("Failure count: " + failureCount.incrementAndGet());

}

}