**ОШ МАМЛЕКЕТТИК УНИВЕРСИТЕТИНИН ЖАРЧЫСЫ**

ВЕСТНИК ОШСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА

BULLETIN OF OSH STATE UNIVERSITY

**ISSN: 1694-7452 e-ISSN: 1694-8610**

№4/2025, 199-211

***ИНФОРМАТИКА***

**УДК: 004.056**

**DOI:** 10.52754/16948610_2025_4_14

# МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ БЕЗОПАСНОСТИ ВЕБ-ПРИЛОЖЕНИЙ НА DJANGO С АКЦЕНТОМ НА ВЫЯВЛЕНИЕ УЯЗВИМОСТЕЙ БИЗНЕС-ЛОГИКИ

БИЗНЕС-ЛОГИКАНЫН АЛСЫЗДЫКТАРЫН АНЫКТООГО БАСЫМ ЖАСОО МЕНЕН DJANGO ВЕБ-ТИРКЕМЕЛЕРИНИН КООПСУЗДУГУН ТЕСТИРЛӨӨ МЕТОДОЛОГИЯСЫ

A METHODOLOGY FOR SECURITY TESTING OF DJANGO WEB APPLICATIONS WITH AN EMPHASIS ON BUSINESS LOGIC VULNERABILITIES

**Омаралиев Абдималик Чырмашович**
*Омаралиев Абдималик Чырмашович*
*Omaraliev Abdimalik Chyrmashovich*

**к.п.н., доцент, Ошский государственный университет**
*п.и.к., доцент, Ош мамлекеттик университети*
*Candidate of Pedagogical Sciences, Associate Professor, Osh State University*
aomaraliev@oshsu.kg
ORCID: 0009-0000-9214-7488

**Карабаев Самат Эшполотович**
*Карабаев Самат Эшполотович*
*Karabaev Samat Eshpolotovich*

**преподаватель, Ошский государственный университет**
*окутуучу, Ош мамлекеттик университети*
*Lecturer, Osh State University*
skarabaev@oshsu.kg
ORCID: 0009-0006-5926-6040

**Омаралиева Гулбайра Абдималиковна**
*Омаралиева Гулбайра Абдималиковна*
*Omaralieva Gulbaira Abdimalikovna*

**к.ф.-м.н., доцент, Ошский государственный университет**
*ф.-м.и.к., доцент, Ош мамлекеттик университети*
*Candidate of Physico-Mathematical Sciences, Associate Professor, Osh State University*
gulya@oshsu.kg
ORCID: 0009-0004-7806-3690

**Данг Вэньхао**
*Данг Вэньхао*
*Dang Wenhao*

**магистрант, Ош мамлекеттик университети**
*магистрант, Ошский государственный университет*
*Master's student, Osh State University*
1159824742@qq.com

# МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ БЕЗОПАСНОСТИ ВЕБ-ПРИЛОЖЕНИЙ НА DJANGO С АКЦЕНТОМ НА ВЫЯВЛЕНИЕ УЯЗВИМОСТЕЙ БИЗНЕС-ЛОГИКИ

**Аннотация**

Статья предлагает практико-ориентированную методологию тестирования безопасности веб-приложений на Django с особым акцентом на уязвимости бизнес-логики – дефекты, которые возникают не из-за низкоуровневых ошибок реализации, а вследствие нарушений инвариантов предметной области и некорректных рабочих процессов. Формализуется понятие инвариантов доменной логики и показывается, как превращать их в проверяемые «оракулы» для тестов. Предлагается связный процесс: от построения модели угроз и сценариев злоупотребления для конкретного Django-проекта до проектирования тестов на основе состояний и свойств, включая property-based и метаморфическое тестирование. Показано, как использовать инфраструктуру Django (middleware, сигналы, транзакции, permissions, URLConf) для целенаправленной инструментализации, логирования и контроля инвариантов, а также как комбинировать юнит-, интеграционные и API-тесты с динамическим и интерактивным тестированием (DAST/IAST). Вводятся метрики покрытия бизнес-логики и даются шаблоны автоматизации в CI/CD. В качестве ориентировочного кейса рассматривается процесс оформления заказа в интернет-магазине: управление скидками, авторизация на уровне объектов, гонки при списании остатков и обход многошаговых проверок. Методология сопровождается местами для иллюстраций, таблиц и фрагментов кода, что делает материал пригодным для внедрения в реальные команды разработки.

***Ключевые слова:*** безопасность веб-приложений; уязвимости бизнес-логики; объектный уровень разрешений; транзакции и целостность данных; CI/CD для безопасности; DAST/IAST

## *БИЗНЕС-ЛОГИКАНЫН АЛСЫЗДЫКТАРЫН АНЫКТООГО БАСЫМ ЖАСОО МЕНЕН DJANGO ВЕБ-ТИРКЕМЕЛЕРИНИН КООПСУЗДУГУН ТЕСТИРЛӨӨ МЕТОДОЛОГИЯСЫ*

**Аннотация**

Макалада Django веб-тиркемелеринин коопсуздугун тестирлөөнүн практикага багытталган методологиясы сунушталат, бизнес-логикалык алсыздыктарга өзгөчө көңүл бурулат – бул төмөнкү деңгээлдеги ишке ашыруу каталарынан эмес, домендин инварианттарын бузуудан жана туура эмес иштөө процессинен келип чыккан кемчиликтер. Домендин логикалык инварианттарынын концепциясын формалдаштыруу жана аларды тесттер үчүн текшерилүүчү “оракулдарга” кантип айландыруу көрсөтүлгөн. Ыраатту процесс сунушталат: конкреттүү Django долбоору үчүн коркунуч моделин жана кыянаттык менен пайдалануу сценарийлерин түзүүдөн тартып, абалдарга жана касиеттерге негизделген тесттерди долбоорлоого чейин, анын ичинде мүлккө негизделген жана метаморфикалык тестирлөө. Django инфраструктурасын (орто программа, сигналдар, транзакциялар, уруксаттар, URLConf) максаттуу инструменттештирүү, инварианттарды каттоо жана көзөмөлдөө үчүн кантип колдонууну, ошондой эле бирдикти, интеграцияны жана API тесттерин динамикалык жана интерактивдүү тестирлөө (DAST/IAST) менен кантип айкалыштыруу керектиги көрсөтүлгөн. Бизнес-логикалык камтуу метрикалары киргизилген жана CI/CDде автоматташтыруу шаблондору камсыздалган. Мисал катары онлайн дүкөнгө заказ

## *A METHODOLOGY FOR SECURITY TESTING OF DJANGO WEB APPLICATIONS WITH AN EMPHASIS ON BUSINESS LOGIC VULNERABILITIES*

**Abstract**

This paper presents a hands-on methodology for security testing of Django web applications with a dedicated focus on business-logic vulnerabilities–defects rooted not in low-level implementation flaws but in violated domain invariants and misdesigned workflows. We formalize domain invariants and demonstrate how to convert them into executable test oracles. The proposed process spans from threat modeling and abuse-case elicitation tailored to a given Django project to state- and property-based test design, including metamorphic testing. We explain how to instrument Django (middleware, signals, transactions, permissions, URLConf) to log, enforce, and verify invariants, and how to combine unit, integration, and API tests with dynamic and interactive techniques (DAST/IAST). We introduce business-logic coverage metrics (role-endpoint matrices, invariant coverage, and critical-scenario coverage) and provide CI/CD automation patterns. An e-commerce checkout case illustrates discount manipulation, object-level authorization, inventory race conditions, and multi-step validation bypass. Throughout, we mark concrete insertion points for figures, tables, and code listings, enabling seamless adoption by engineering teams.

берүү процесси каралган: арзандатууларды башкаруу, объект деңгээлинде авторизация, балансты өчүрүү жана көп баскычтуу текшерүүлөрдү айланып өтүү. Методология иллюстрациялар, таблицалар жана код фрагменттери үчүн орундар менен коштолот, бул материалды иштеп чыгуу командаларында ишке ашырууга ылайыктуу кылат.

***Ачкыч сөздөр:*** веб-тиркеме коопсуздугу; бизнес-логикасынын кемчиликтери; объект деңгээлиндеги уруксаттар; метаморфикалык тестирлөө; транзакциялар жана маалыматтардын бүтүндүгү; коопсуздук үчүн CI/CD; DAST/IAST;

***Keywords:*** web application security; business logic vulnerabilities; object-level permissions; transactions and data integrity; CI/CD for security; DAST/IAST

**Введение**

Безопасность веб-приложений часто сводят к «технике»: инъекции, XSS, конфигурация. Однако самые дорогие инциденты всё чаще происходят на уровне бизнес-логики – там, где нарушаются инварианты предметной области и последовательности действий. Классические сканеры тут мало помогают, потому что дефект скрыт не в строке кода, а в неправильном процессе: скидки суммируются сверх правил, пользователь меняет адрес после оплаты, остатки списываются из параллельных запросов, объектные разрешения обходятся фильтрами.

Django создаёт ощущение защищённости «батарейками из коробки»: аутентификация, middleware, ORM и транзакции, DRF, админка. На практике проверки легко размазываются по слоям – часть в сериализаторе, часть во view, часть в моделях и фоновых задачах, плюс внешние вебхуки. Результат – трещины на стыках состояний и ролей: IDOR на уровне объектов, расхождение кэша и БД, обход многошаговых проверок в «мастерах» и чек-аутах. В этой работе мы предлагаем воспроизводимую методологию тестирования безопасности Django-приложений именно на уровне бизнес-логики. Ключевая идея – явно формулировать доменные инварианты и превращать их в исполняемые тест-оракулы. Демонстрируется как от компактной модели угроз и злоупотреблений перейти к тестам, основанным на состояниях и свойствах, как целенаправленно инструментировать слои Django и как измерять не «покрытие строк», а покрытие инвариантов и матрицы ролей эндпоинтов. Практическая ценность методологии – в едином языке между разработчиками, аналитиками и тестировщиками, в генеративном характере проверок, который находит неожиданные комбинации состояний, и во встроенности в CI/CD, где регрессии логики ловятся до релиза. В качестве сквозного примера используется типичный интернет-магазин: корзина, скидки, оплата, доставка, возвраты – достаточный по сложности контур, чтобы рассмотреть объектные разрешения, денежные инварианты, гонки и многошаговые процессы.

Далее кратко разбераются архитектурные точки риска в Django, формализуруются инварианты и связь их с таксономией уязвимостей бизнес-логики, затем идет переход к поэтапной методологии проектирования тестов, подготовке данных и обеспечению изолированности, обсуждаются сложные сценарии конкуренции и работы с деньгами, интеграцию с DAST/IAST, метрики эффективности и их автоматизацию в CI/CD. Завершаются кейс-стади и ограничениями подхода.

**Обзор литературы**

Проблематика уязвимостей бизнес-логики в веб-приложениях получила активное развитие в последние годы как в зарубежной, так и в отечественной литературе. Большинство работ исходят из признания того факта, что традиционные механизмы анализа безопасности не охватывают классы дефектов, возникающие из-за нарушений предметных правил и состояний процессов.

Значительный вклад в формализацию подходов внёс проект OWASP. В Application Security Verification Standard (ASVS) и в Web Security Testing Guide выделяются специальные разделы, посвящённые тестированию бизнес-логики. Подчёркивается, что именно эти уязвимости труднее всего обнаруживаются автоматизированными средствами и требуют понимания сценариев использования приложения. Дополнительным шагом стало появление

инициативы OWASP Business Logic Abuse Top 10, где собраны наиболее частые злоупотребления рабочими процессами. В англоязычной академической литературе можно выделить работу Ким и соавторов, где предложена методика поиска манипуляций бизнес-потоками на клиентской стороне (Kim, 2020). Совсем недавно Tunalı и Kepir опубликовали обзор с применением искусственного интеллекта для выявления логических уязвимостей. Эти публикации фиксируют важную тенденцию: от ручного анализа к формализации свойств и автоматизации поиска нарушений (Tunalı, 2025).

В российском и кыргызском научном пространстве проблематика также отражена. Статья Карабаева С. Э. акцентирует внимание на сравнительном анализе архитектурных паттернов Django и ASP.Net Core, указывая на влияние структурных решений на устойчивость бизнес-логики (Карабаев, 2025). Работа Путято и коллег рассматривает типовые уязвимости веб-приложений в целом, в том числе затрагивая бизнес-правила и вопросы целостности данных (Путято, 2022). Популярные изложения для разработчиков – статьи на VC.ru, Habr и выступления на конференциях (Analyst Days) – делают акцент на прикладных примерах: скидки, многопользовательские операции, гонки при резервировании.

Фреймворк Django имеет официальное руководство по безопасности, где систематизированы меры по защите от технических атак, однако отдельного раздела про бизнес-логику нет. Тем не менее встроенные механизмы – транзакции, ограничения БД, middleware, object-level permissions в Django REST Framework – являются основой для реализации проверок инвариантов. Отдельного внимания заслуживает библиотека Hypothesis, позволяющая задавать свойства и проверять их на широком пространстве входов, что соответствует научным подходам property-based тестирования.

Таким образом, литература фиксирует сдвиг от рассмотрения исключительно технических уязвимостей к изучению бизнес-логики как самостоятельного класса угроз. В англоязычных источниках преобладают формальные и автоматизированные методы поиска, в русскоязычных – прикладные примеры и архитектурные аспекты. Обе линии сходятся в признании необходимости формализации доменных инвариантов и их систематической проверки в ходе разработки и эксплуатации веб-приложений. Именно на этой основе выстраивается предлагаемая в статье методология.

**Инварианты, тест-оракулы и злоупотребления**

Начинать стоит не с списка уязвимостей, а с вещей, которые никогда не должны ломаться в вашем бизнесе. Это и есть инварианты. В интернет-магазине их немного, но каждый – как страховочный трос: сумма заказа не уходит в минус; скидка не больше подытога; клиент видит только свои объекты; возвраты не превосходят оплаченного; остатки не становятся отрицательными даже под параллельной нагрузкой. Если эти фразы остаются только «в голове у команды», они сгорают в ночных релизах. Их надо зашить в систему так, чтобы они проверялись автоматически – и в тестах, и в проде.

Технический инструмент для этого прост: превращаем каждую фразу в исполняемую проверку – оракул. Где именно проверять – зависит от характера инварианта. Денежный баланс и остатки лучше фиксировать ближе к данным: ограничениями БД и доменным слоем, который вызывается из всех путей – и из API, и из админки, и из Celery. Авторизацию, наоборот, удерживаем там, где формируется выборка и доступ к объекту.

Никаких «магических» проверок в одном-единственном сериализаторе – фоновая задача их обойдет за пять минут.

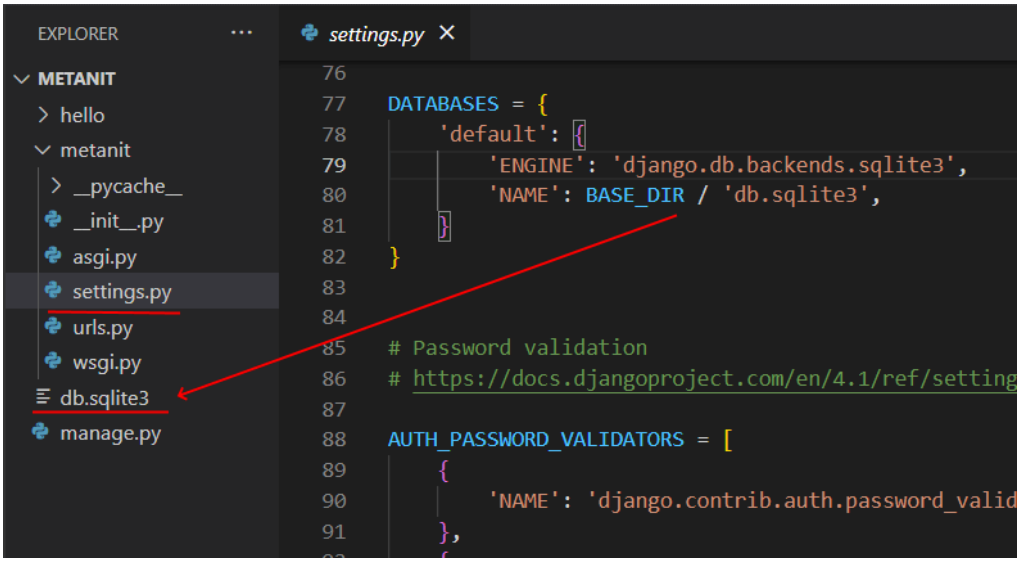


Рисунок 1. Потоки данных и доверительные границы в Django-приложении

*from django.db import models*

*from django.db.models import Q, F*

*class Order(models.Model):*

*subtotal = models.DecimalField(max_digits=12, decimal_places=2)*

*discount = models.DecimalField(max_digits=12, decimal_places=2, default=0)*

*total = models.DecimalField(max_digits=12, decimal_places=2)*

*class Meta:*

*constraints = [*

*models.CheckConstraint(check=Q(total__gte=0), name="order_total_non_negative"),*

*models.CheckConstraint(check=Q(total__lte=F("subtotal")),*
*name="order_total_le_subtotal"),*

*]*

А в доменном слое просто одна функция, которую вызывают все:

*from decimal import Decimal*

*def recompute_total(subtotal: Decimal, discount: Decimal) -> Decimal:*

*total = (subtotal - discount).quantize(Decimal("0.01"))*

*assert Decimal("0.00") <= total <= subtotal, "discount invariant"*

*return total*

Именно на этом уровне удобно писать тест-оракулы свойств. Не «если купон такой-то, ожидаем X», а «какой бы ни был купон, итог не уходит ниже нуля и не превышает

подытог; повторное применение купона не меняет результат». Такой тест переживет смену правил, потому что проверяет не кейс, а смысл. Минимальный пример на Hypothesis у нас уже намечен во введении – дальше мы просто подменим фиктивные фабрики реальными.

С авторизацией картина другая. Инвариант звучит так: пользователь видит только свои объекты. Это не про «IsAuthenticated», это про область видимости выборки и проверку на уровне объекта. В DRF это решается скупо, но надёжно: queryset сразу ограничивается пользователем, а get_object не обходит этот фильтр. Никаких .get(pk=...) поверх сырой модели. Как только вы поймали себя на мысли «ну тут-то можно», вспомните, что любая поблажка станет входом для IDOR.

Состояния – отдельная тема. У заказа есть жизнь: created → paid → shipped → returned. Многие дефекты рождаются именно на переходах. Инвариант тут формулируется по-честному: после оплаты адрес менять нельзя; возврат не может превысить оплаченного; списания остатков происходят до отправки. Технически это удобно ловить через целевые контроллеры переходов или метод-команду на модели, а не россыпь if в разных местах. Пример из реальной жизни: в обработчике вебхука вы всегда рискуете получить одну и ту же нотификацию дважды. Если переход «оплачено» не идемпотентен, вы удвоите деньги и нарушите инвариант. Лекарство стандартное: уникальный маркер события и блокировка на время обновления.

```
from django.db import transaction, IntegrityError
from .models import PaymentEvent, Payment

def apply_payment(provider_event_id: str, order_id: int, amount: int):
    with transaction.atomic():
        try:
            PaymentEvent.objects.create(provider_event_id=provider_event_id)
        except IntegrityError:
            return  # повтор – тихо выходим, инвариант сохранён

        pay, _ = Payment.objects.select_for_update().get_or_create(order_id=order_id)
        pay.captured += amount
        pay.save(update_fields=["captured"])
```

Злоупотребления – это не «хитрые хакеры делают», это просто последовательности действий, которые нарушают инварианты. Клиент кладёт товар в корзину, открывает два окна, в одном применяет купон, в другом – тоже, оплачивает по старой цене и ловит рассинхрон. Оператор в бэкофисе меняет статус, не видя, что по вебхуку ещё идёт подтверждение. Партнёр редактирует заказ маркетплейса и незаметно сдвигает границы арендатора. Хорошая новость: у каждого такого сценария есть очень конкретная проверка.

«Сумма скидок не больше подытога»; «после оплаты адрес недоступен к редактированию»; «в запросе на детализацию заказа tenant_id совпадает с tenant_id пользователя»; «резерв остатка делается под select_for_update и никогда не уходит ниже нуля». Чем точнее вы сформулируете эти фразы, тем проще написать тест, который будет жить годами.

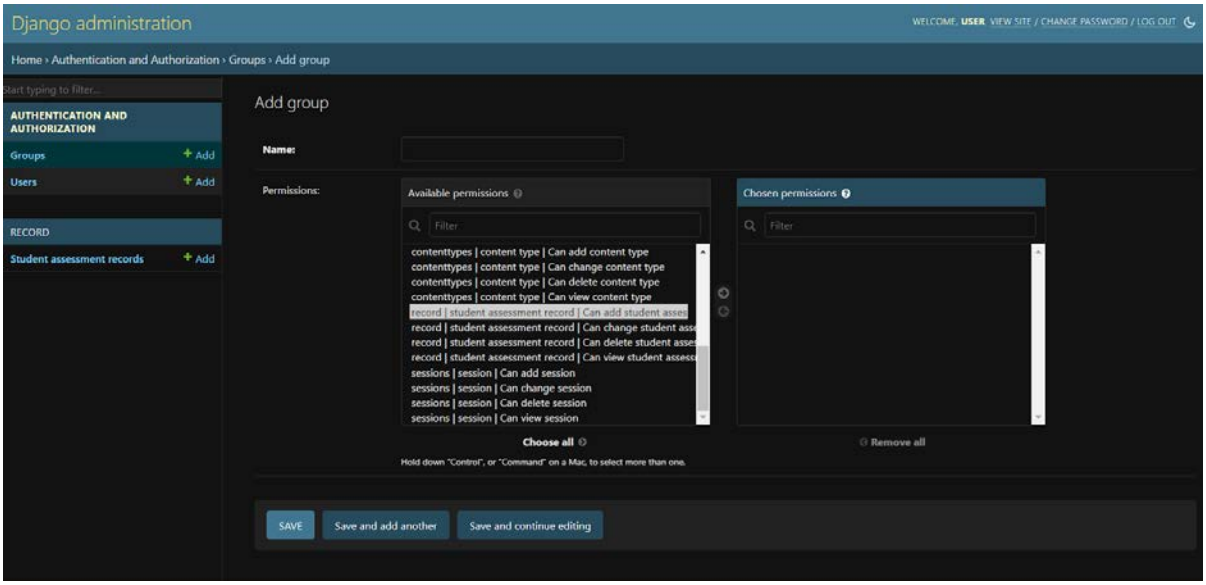


Рисунок 2. Матрица доступа ролей к операциям API

Что важно сделать до любых «сканеров» и «пайплайнов». Во-первых, выписать три-пять главных инвариантов обычными предложениями, без техно-слов. Во-вторых, для каждого приготовить одно место в системе, где он будет проверяться «железно»: либо ограничение БД, либо доменная функция, либо контроллер переходов. В-третьих, добавить тонкий слой наблюдаемости: корреляционный идентификатор запроса, актор, состояние до и после. Когда тест внезапно поймает нарушение, показывается не скриншот логов отчаяния, а аккуратную запись: «actor=alice, endpoint=orders.apply_coupon, was=created, became=created, subtotal=1000, discount=1200 – invariant breach».

Когда вся эта конструкция появляется, тест-дизайн становится почти механическим. Берется инвариант «остаток не уходит ниже нуля», прогоняется два параллельных запроса на резерв одного и того же SKU и проверяется, что один из них падает контролируемым способом, а не помпой исключений посреди транзакции. Берется инвариант «после оплаты адрес редактировать нельзя» и формулируется свойство: в любых данных, где paid_at уже установлен, PATCH на адрес возвращает 403 и не изменяет запись. Инвариант «пользователь видит только своё» и по матрице ролей генерируется набор API-вызовов, в которых токен «чужого» пользователя никогда не вытаскивает объект за границей его QuerySet.

Формализация инвариантов и уязвимостей бизнес-логики

Под бизнес-логикой понимается совокупность правил предметной области, реализованных на уровне доменных моделей и их переходов. Уязвимость бизнес-логики – это достижимое исполнением нарушение доменного инварианта при корректно сформированных с точки зрения протокола запросах.

Операционализация. Чтобы инварианты были проверяемыми, их фиксируют одновременно на трёх уровнях: (i) база данных (ограничения и ключи), (ii) доменный код

(функции-команды/агрегаты), (iii) точки входа (контролёры/серилизаторы/ORM-скоупинг). Это даёт баланс «звуковости» и наблюдаемости: нарушение либо невозможно конструктивно, либо регистрируется немедленно.

```python
from rest_framework.viewsets import ModelViewSet
from rest_framework.permissions import IsAuthenticated, BasePermission
from .models import Order
from .serializers import OrderSerializer

class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.user_id == request.user.id

class OrderViewSet(ModelViewSet):
    serializer_class  = OrderSerializer
    permission_classes = [IsAuthenticated, IsOwner]

    def get_queryset(self):
        return Order.objects.filter(user=self.request.user)

    def get_object(self):
        # критично брать объект из уже «сузленного» queryset
        return self.get_queryset().get(pk=self.kwargs["pk"])
```

Рисунок 3. Существенные фрагменты для денежных инвариантов

В научных терминах оракулом теста выступает вычислимая аппроксимация на наблюдаемом фрагменте состояния. Для сложных систем эффективны свойства, инвариантные относительно ранжирования сценариев, а также метаморфические отношения между входами и выходами.

Таблица 1. Классы инвариантов и уровни их контроля в Django

| **Класс инварианта** | **Пример формулировки** | **Где фиксировать в первую очередь** | **Что проверяет тест** |
|---|---|---|---|
| Денежные | 0≤total≤subtotal0 \leq \texttt{total} \leq \texttt{subtotal} | CheckConstraint + доменная функция перерасчёта | равенство расчётов при любых допустимых данных; отсутствие отрицательных итогов |
| Остатки/квоты | stock≥0 \texttt{stock} \ge 0 при любых гонках | select_for_update в транзакции | параллельные резервы не ведут к stock<0stock<0 |
| Доступ к объектам | субъект видит только свои объекты | scoping QuerySet + object-level permission | выборка чужого pk не достижима; 403 вместо данных |
| Состояния | запрещённые | контроллер | PATCH после оплаты |

| | | | |
|---|---|---|---|
| процессов | переходы отсутствуют | переходов/команда модели | не меняет адрес |
| Идемпотентность внешних событий | повторный вебхук не удваивает эффект | уникальный индекс события + транзакция | два одинаковых события эквивалентны одному |
| Межарендаторная изоляция | tenant-идентичность сохраняется | ключи и фильтры по tenant_id на всех слоях | невозможна «протечка» данных между арендаторами |

**Методология: как превратить доменные инварианты в проверяемые свойства**

Тестирование бизнес-логики имеет смысл лишь тогда, когда инварианты предметной области формулируются как проверяемые свойства, привязанные к конкретным слоям системы. В контексте Django естественно закреплять один и тот же инвариант одновременно в базе данных, в доменном коде и на уровне точек входа. Такой трёхслойный контроль устраняет «щели» между путями исполнения и исключает зависимость от одного-единственного валидатора. Под инвариантом будем понимать предикат $I(s)$I(s), истинный для всех допустимых состояний $s$s. Оракул теста – вычислимая аппроксимация $I$I на наблюдаемом артефакте: ответе API, записи в БД, событии журнала. Для денежных расчётов это, например, ограниченность итога. В Django она фиксируется как CheckConstraint в модели заказа, как чистая функция перерасчёта в доменном слое и как пост-проверка в контроллере. Для остатков – недопустимость отрицательных значений при конкуренции: в БД – инвариант области значений, в домене – операция под транзакцией, в ORM – блокировка select_for_update на критичном агрегате.

Генеративное тестирование оперирует свойствами, а не наборами кейсов. Для скидок работают свойства ограниченности, монотонности и идемпотентности применения; для внешних событий – идемпотентность по уникальному идентификатору и «ровность» эффекта при повторе. Для доступа к объектам важен инвариант области видимости: субъект не может наблюдать и модифицировать чужие объекты. В терминах DRF это означает, что выборка формируется «узким» get_queryset() с учётом субъекта, а получение конкретного экземпляра не обходит этот скоуп. В правильно спроектированном API попытка обратиться к чужому объекту приводит к «исчезновению» последнего из области видимости (404), а не к запрету операции на уже раскрытом факте (403). Состояния и переходы – главный источник логических дефектов. После оплаты адрес доставки становится неизменяемым; возврат средств не может превышать фактическую сумму зачислений; списание остатков допустимо только до отгрузки. Эти правила лучше выражать не россыпью условных проверок, а узкими командами переходов, чтобы один и тот же интерфейс использовался из REST, админки и фоновых задач. Тогда свойство «адрес неизменяем после оплаты» проверяется одинаково для всех путей, а нарушение не может «просочиться» через вебхук или Celery-таск. Наблюдаемость – обязательный компонент методологии. Любая проверка инварианта должна сопровождаться контекстом расследования: корреляционный идентификатор запроса, субъект действия, прежнее и новое состояние доменного объекта, существенные числовые значения. Достаточен лёгкий протокол безопасности в журнале приложения; этого

хватает, чтобы превращать тесты в воспроизводимые эксперименты и связывать обнаружение нарушения с конкретной траекторией исполнения. Данные для проверок должны быть реалистичными и контролируемыми. Фабрики и сид-снимки позволяют воспроизводить состояния без побочных эффектов между тестами; генераторы входов – варьировать значимые параметры в широкой области определений. Важно избегать «флейков» при конкурентных сценариях: тесты, моделирующие параллельные операции, выполняются в отдельных транзакциях, а доменные операции обязаны удерживать блокировки на время изменений, чтобы отрицательные остатки и «двойные» платежи были конструктивно недостижимы, а не просто «не воспроизводились на стенде».

Метрики эффективности выходят за пределы традиционного «покрытия строк». Содержательными оказываются три группы: доля инвариантов, имеющих проверку на всех трёх слоях; полнота матрицы ролей и эндпоинтов (включая негативные траектории «чужой объект»); охват критичных сценариев бизнес-процесса – тех, где сочетание ролей и переходов приводит к максимальной экономической тяжести ошибки. Эти метрики интегрируются в CI таким образом, чтобы регресс любого инварианта делал релиз блокирующим, а отчёт содержал контекст нарушения из журнала безопасности.

Такое изложение сознательно минималистично в коде: акцент смещён на формальные свойства и точки крепления в архитектуре Django. В следующих разделах можно локально углубляться – например, показать один концентрированный пример идемпотентного обработчика вебхуков или короткий тест на конкурентный резерв остатков – но только там, где пример действительно проясняет формулировку свойства, а не дублирует её в синтаксисе.

**Заключение**

Представленная работа показала, что уязвимости бизнес-логики в веб-приложениях не могут рассматриваться как побочный эффект реализации, а должны трактоваться как нарушения фундаментальных свойств предметной области. Именно поэтому ключевым элементом методологии стало формальное закрепление инвариантов и их последующее преобразование в исполняемые тестовые оракулы. Такой подход делает возможным не просто «ловить ошибки», но обеспечивать воспроизводимый контроль соблюдения правил бизнеса на всех уровнях системы.

Django оказался удобной платформой для внедрения этой практики. Слои фреймворка – модели, сериализаторы, middleware, транзакции и сигналы – предоставляют естественные точки крепления для проверки инвариантов. Совмещение этих механизмов с property-based и метаморфическим тестированием позволяет выходить за пределы ручных сценариев и проверять устойчивость логики в широких пространствах состояний. В результате тестирование превращается в исследование допустимых свойств системы, а не в перебор ожидаемых кейсов. Особое внимание было уделено конкуренции, идемпотентности внешних событий и матрице ролей. Эти аспекты остаются источниками наиболее дорогих дефектов, и именно их автоматическая проверка даёт наибольший эффект для команд, работающих в высоконагруженных и финансово чувствительных доменах. Интеграция в CI/CD позволяет закрепить этот контроль как непрерывный процесс, где нарушение инварианта воспринимается так же критично, как падение юнит-теста или деградация производительности.

Ограничения подхода связаны прежде всего с архитектурными особенностями: распределённые микросервисы, GraphQL-интерфейсы и стриминговые протоколы требуют дополнительных средств формализации. Тем не менее базовый принцип остаётся универсальным: безопасность бизнес-логики начинается с чёткой формулировки того, что должно быть истинным всегда, и заканчивается автоматизированной проверкой этих утверждений в коде и данных.

Таким образом, методология превращает инварианты предметной области в полноценный инструмент инженерного контроля. Она создаёт мост между бизнес-правилами и техническими артефактами, делая тестирование не вспомогательным процессом, а частью научно обоснованной практики обеспечения безопасности веб-приложений.

**Список литературы**

1. Аркабаев, Н., Алымова, З. (2024). Разработка WEB серверных приложений на базе .net core в примере интернет-магазина. *Вестник Ошского государственного университета*, (1), 142–154. https://doi.org/10.52754/16948610_2024_1_13

2. Аркабаев, Н., Арапбаев, Б., Ражапов, С. (2025). Трансформация финансового анализа с помощью языка программирования python: от рутинных вычислений к стратегическим решениям. *Вестник Ошского государственного университета*, (2), 122–132. https://doi.org/10.52754/16948610_2025_2_11

3. Карабаев, С.Э. (2025). Django vs ASP.Net Core: сравнительный анализ архитектурных паттернов для масштабируемых enterprise-решений. *Research and Implementation*, 3(3), 68–77. https://zenodo.org/records/15019033

4. Kirill. (2025). Безопасность Django-приложений: главные угрозы и защита на практике. VC.ru. https://vc.ru/dev/2124507-bezopasnost-django-prilozheniy-ugrozy-zashchita

5. Омаралиева, Г., Абдумиталип уулу, К., Жунусова, Г., Санжар кызы, А. (2025). Сравнительный анализ мотивации к изучению программирования у студентов направлений ИВТ и ИСТ института МФТИТ ОШГУ. *Вестник Ошского государственного университета*, (2), 81–93. https://doi.org/10.52754/16948610_2025_2_8

6. Орликов, В. (2025). «Казнить нельзя помиловать»: уязвимости из-за ошибок в бизнес-логике. Analyst Days. https://analystdays.ru/ru/talk/117628

7. Путято, М.М., Макарян, А.С., Лещенко, В.В., Немчинова, В.О. (2022). Анализ типовых уязвимостей при построении веб-приложений. *Информатика и системы управления,* (3), 306–315.

8. Тажикбаева, С.Т., Айтбай кызы, Д., Тагаева, Г.Т. (2025). Python программалоо тилинин мүмкүнчүлүктөрүн окутуу процессинде колдонуу. In *Билим берүүдөгү математика, физика жана маалыматтык технологиялардын актуалдуу маселелери* (pp. 144–149).

9. Hypothesis. (n.d.). Hypothesis – property-based testing for Python. https://hypothesis.works/

10. Kim, I.L., Zheng, Y., Park, H., Wang, W., You, W., Aafer, Y., Zhang, X. (2020). Finding client-side business flow tampering vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE ’20)* (pp. 1–12).

11. Sane, P. (2020). *Is the OWASP Top 10 list comprehensive enough for writing secure code?* arXiv. https://arxiv.org/abs/2002.11269

12. Tunalı, A., Kepir, Y. (2025). Business logic vulnerabilities in the digital era: A detection framework using artificial intelligence. *Information*, 16(7), Article 585. https://doi.org/10.3390/info16070585