МИР Plat.Form

Как устроена самая современная платежная система в МИРе

МИР Plat.Form
Как устроена самая современная платежная система в МИРе

СОДЕРЖАНИЕ

ВСТУПЛЕНИЕ К КНИГЕ ОТ ЛИЦА ЗАМЕСТИТЕЛЯ ГЕНЕРАЛЬНОГО ДИРЕКТОРА НАЦИОНАЛЬНОЙ СИСТЕМЫ ПЛАТЕЖНЫХ КАРТ (НСПК) ВЛАДИМИРА ТРОЯНОВСКОГО

Вы держите в руках книгу, в которой нам удалось подвести итог 6-летней работы над платежной системой «Мир». Быть первопроходцем всегда непросто. Но, несмотря на все сложности, нам удалось в кратчайшие сроки реализовать уникальный проект, создать с нуля отечественную IT-инфраструктуру, аналогов которой не было в России.

Вернемся к истокам. Национальная система платежных карт (НСПК) была создана весной 2014 года, когда международные платежные системы отключили несколько российских банков. Я с большим теплом и гордостью вспоминаю начало этого большого пути: в нашей команде было всего 10 человек, включая генерального директора, а перед нами стояла сложная задача, которую нужно было решить в очень сжатые сроки. Мы закладывали надежные, простые и взаимозаменяемые решения, благодаря чему с 1 апреля 2015 года межбанковские платежи внутри России по всем пластиковым картам стали проходить через Национальную систему платежных карт.

Эмиссия карт «Мир» не заставила себя долго ждать — первые банки присоединились к системе уже в декабре 2015 года. Создание новых продуктов, их развитие и поддержка требовали огромных усилий множества профессионалов. Именно поэтому в 2020 году появился бренд Мир Plat.Form, который объединил всех наших IT-специалистов в одну команду.

Мир Plat.Form — это команда профессионалов, которые создают многочисленные технологические решения Национальной системы платежных карт. Каждый день мы обеспечиваем бесперебойность и доступность операций по картам всех международных платежных систем в России, а также платежной системы «Мир». Кроме того, Мир Plat.Form обеспечивает операционную функцию для Системы быстрых платежей и работает над созданием инновационных платежных решений с использованием современной инфраструктуры и ведущих мировых подходов к разработке.

Благодаря слаженной работе команды Мир Plat.Form, в 2021 году нам удавалось обеспечивать 2500 транзакций в секунду при отказоустойчивости и доступности 99,999%. Мы развиваем российскую финтех-индустрию в тесной коммуникации с российскими финансовыми институтами, чтобы миллионы наших соотечественников распоряжались своими деньгами максимально просто, быстро и безопасно.

В книге «Как устроена самая современная платежная система в МИРе» мы поделились нашим опытом, успешными кейсами, советами по оптимизации процессов и сохранению максимальной надежности и доступности продуктов платежной системы «Мир».

Популярность IT-решений Национальной системы платежных карт и количество пользователей наших сервисов постоянно растет: по итогам 9 месяцев 2021 года выпущено 108,6 млн карт «Мир», на нее приходится 5,2% всех операций по картам в России и 32,3% выпуска новых карт, и я убежден, что это только начало большого пути.

О ЧЕМ И ДЛЯ КОГО ЭТА КНИГА

В наши дни происходит стремительный рост в области информационных технологий. С завидной регулярностью возникают IT-стартапы, которые меняют нашу привычную жизнь. Мы каждый день пользуемся сервисами, которых несколько лет назад еще не существовало, и сама наша жизнь меняется.

Перед вами книга, подготовленная специалистами Мир Plat.Form совместно с экспертами экосистемы HighLoad++.

> **Мир Plat.Form** — это IT-бренд, объединяющий технологические инструменты и сервисы Национальной системы платежных карт. Команда разрабатывает и поддерживает платформенные сервисы и решения в разных направлениях финтеха с использованием современной технологической инфраструктуры и лучших мировых подходов к разработке.
>
> **Мир Plat.Form** параллельно ведет несколько значимых IT-проектов: обеспечивает бесперебойность и доступность операций по картам всех платежных систем в России, а также самой платежной системы «Мир». Кроме того, Мир Plat.Form обеспечивает операционную функцию для Системы быстрых платежей и работает над созданием инновационных платежных решений.

Мы хотим рассказать о том, как устроена и работает платежная система — огромный проект, результаты которого ежедневно используют миллионы людей. Книга будет интересна айтишникам и не только — всем, кто хочет узнать, как работает софт, который может помочь в любую секунду провести платеж по карте «Мир».

К платежной системе, как к IT-проекту, предъявляются повышенные требования: отказоустойчивость, надежность, скорость отклика, сохранность персональных и финансовых

данных. Как эти требования влияют на архитектуру? Почему мы должны выбирать те или иные инструменты и как можем их использовать?

Мы с вами пройдем по всем направлениям разработки и поймем принципы, которые транслируются, — от сверхтребований к проекту до ежедневной деятельности разработчиков.

АРХИТЕКТУРНЫЙ ПЛАН ПЛАТЕЖНОЙ СИСТЕМЫ «МИР»

Рост объема обрабатываемых данных, интенсивность генерации трафика в интернет-среде, новые сценарии использования информации — все это меняет архитектурные подходы, применяемые в информационных системах. В данной главе будут описаны основные моменты, связанные с задачами, которые решает современная информационная система, а также пойдет речь о том, как ее сделать надежной, быстрой и отзывчивой.

Начнем мы, конечно, с общих архитектурных принципов. Рассмотрим понятие архитектуры, изучим, как происходит выбор тех или иных архитектурных паттернов, а затем попробуем сделать проброс от задач к архитектурным решениям. Ну и конечно, поговорим о том, как платформа обеспечивает заветные пять девяток.

ЧТО ТАКОЕ АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Как можно описать информационную систему? Обыватель, возможно, представит что-то нематериальное, где есть нолики и единички, как нас учили на уроках информатики. Вполне возможно, кто-то назовет слово «база», подразумевая, что обычно идет речь об информации и ее хранении.

Как нолики и единички превращаются в изображение на мониторе или в текстовый файл, который вы набираете в редакторе? Айтишник скажет: «Ну как же, все просто: процессор обрабатывает посылаемые сигналы из подсистемы ввода/вывода, записывает в оперативную память, отправляет в шину обмена данными, откуда видеокарта читает, преобразовывает в видеосигнал, попутно пересчитав координаты точек в

полигоны (или наоборот) и записав сигнал во входящий порт монитора».

Все подобные описания — и на языке обывателя, и в технических терминах — и составляют суть архитектуры информационных систем.

Архитектура информационной системы, или IT-архитектура — это совокупность функций системы, ее компонентов и методов их взаимодействия между собой, а также с другими информационными системами, пользователями и другими акторами.

Для чего нужно описывать IT-архитектуру? Прежде всего, это необходимо для общего понимания задач, решаемых системой, и подходов, которых нужно придерживаться при их реализации. Также следует знать, что IT-архитектура включает в себя функциональную, компонентную и интеграционную составляющие.

- Функциональная архитектура представляет собой перечень функций, выполняемых системой, сценариев поведения системы и ее пользователей в различных состояниях.
- Компонентная архитектура перечисляет и описывает составные части системы и то, какие функции каждый из этих компонентов выполняет. Компоненты системы следует воспринимать на разных уровнях абстракции:

    1. Уровень инфраструктуры, железа. Отвечает на вопрос, какое оборудование понадобится для работы системы и как это оборудование должно быть связано в вычислительную сеть. Условно это можно представить на схеме, где указаны серверы, их характеристики (CPU, RAM, HDD и т.д.), IP-адреса и другая полезная информация, а также сетевые связи между серверами (IP-адрес источника, IP-адрес назначения, TCP/UDP-порт и др.). Также на этом уровне присутствует схема коммутации серверного оборудования (в ней можно найти, каким кабелем какой порт коммутатора подсо-

единен к какому порту сервера или к другому оборудованию) и т.д.
2. Уровень прикладного ПО. Показывает взаимодействие прикладных программных компонентов между собой с привязкой к серверному оборудованию. Схематично представляется как набор системных программных компонентов (ОС, сервер приложений, СУБД и др.) и поименованного ПО, исполняемого в системных компонентах (инстанс БД, JAR-файл, Docker-контейнер и т.д.). Схематично между компонентами также стоит указывать взаимосвязи с определенным прикладным протоколом (6, 7 уровни модели OSI, например, ISO8583, HTTPS).
3. Уровень разработки. При написании кода приложения также формируется его архитектура, состоящая из объектов, бизнес-логики, адаптеров, фреймворков и др. Здесь тоже может быть представлено несколько уровней абстракции, от прямой работы с памятью до использования готовых библиотек.

- Интеграционная архитектура описывает то, какие данные система и ее компоненты получают и передают, а также какие протоколы передачи данных должны использоваться при этих взаимодействиях. Важно обозначить источник и инициатора передачи данных, т.к. в последующем это поможет определить схему сетевого взаимодействия оборудования.

Что отличает архитектуру информационной системы от, скажем, архитектуры здания? Наибольшее отличие в том, что архитектура здания — это структура, которая полностью рассчитана, описана, оцифрована до начала строительства. А вот при разработке информационных систем каждый шаг разработчика — это некое архитектурное решение. Ведь в здании узлы и конструкции рассчитаны заранее и именно они определяют итоговые характеристики (вернее, подобраны для заданных условий). А в информационных технологиях нет одного решения частной задачи и разработчик ежеминутно сталкивается с необходимостью решить — «приклеить или прибить?». Поэтому и архитектура информационной системы постоянно развивается и эволюционирует, получает новые свойства и детализируется с каждой новой строчкой кода.

И еще немного терминологии. Архитектурное решение — это проекция архитектуры информационной системы на текущий момент, описание желаемого будущего состояния архитектуры системы и перечень изменений, которые требуются для перехода из первого состояния во второе. И все это в терминах функциональной, компонентной и интеграционной архитектур.

ДЛЯ ЧЕГО НЕОБХОДИМО АРХИТЕКТУРНОЕ РЕШЕНИЕ

Разработка программного обеспечения как проект — это часто длинный путь. От концепта, так называемого MVP (минимально жизнеспособный продукт) до продуктового решения, которое постоянно наращивает функциональность с каждым релизом. На первом этапе, вероятно, требуется оценка стоимости разработки «сверху». Здесь архитектурное решение является экспертной оценкой будущего проекта, основывается на знаниях рабочей группы и ограничений компании или проекта.

Что имеется в виду под ограничениями:

1. Установленный в компании перечень используемых технологий (например, решено использовать только СУБД MySQL или только виртуальные машины KVM).
2. Ограниченное время на разработку системы (например, нужно запустить MVP раньше конкурента, желательно прямо сейчас).
3. Финансовый лимит, когда на проект выделена определенная сумма.

Архитектура ответит на вопрос «сколько вешать в граммах»: какие вычислительные ресурсы понадобятся (какие «ядра — чистый изумруд» нам надо арендовать, сколько оперативной

памяти и дискового хранилища утилизируем на требуемом горизонте времени), какие лицензии на программное обеспечение потребуется приобрести, какие программные компоненты надо будет разработать самостоятельно. Либо нам нужно для MVP всего лишь «запилить сайтик» на Tilda и прикрутить к нему табличку в GoogleDocs.

Архитектурный процесс запускает постоянную адаптацию и детализацию архитектуры информационной системы, когда каждая вновь появляющаяся ее часть позволяет более точно определить состояние «as is» и «to be» (т.е. то, какая система сейчас, и то, какой она будет в определенной целевой точке). Процесс похож на игру «Змейка», когда с каждой новой «съеденной» точкой змея становится больше (для архитектуры «больше» — не характеристика размера, а, скорее, увеличение точности, ясности): точнее становится функциональная архитектура, больше точности в реализации компонентов и т.д.

Есть смысл полагать, что самое важное свойство архитектурного решения в том, что оно дает понять разным людям, что им понадобится сделать в проекте: программисту — какой код написать, инфраструктуре — какие серверы купить, менеджерам — какой бюджет запланировать и т.д.

НА КАКИХ ПРИНЦИПАХ ОСНОВЫВАЕТСЯ ВЫБОР АРХИТЕКТУРНЫХ РЕШЕНИЙ

Не секрет, что одну работу можно выполнить по-разному. И даже иногда при этом получить одинаковый результат. В информационных технологиях проблема выбора инструментов стоит особенно остро. Под инструментом понимается и язык разработки ПО, и вспомогательные системы, такие как хранилища данных, серверное оборудование, операционные системы и др.

Как выбрать из многообразия вариантов оптимальный? В чем будет его оптимальность? Однозначного ответа на этот вопрос

не будет, но есть подсказка: **ВДК** — время, деньги, качество. В каждом случае выбирайте актуальные два критерия из трех. А дальше детализируйте их применительно к конкретной системе или проекту.

Ниже вы можете увидеть пример так называемой пирамиды оценки архитектурных решений одной из систем Мир Plat.Form.

Tier 1. Критичные параметры, несоблюдение которых может привести к остановке системы, остановке разработки системы или серьезно повысит эти риски.

Критичные параметры, несоблюдение которых может привести к остановке системы, остановке разработки системы или серьезно повысит эти риски

1. Время разработки (шкала оценки «чем быстрее до прода, тем лучше»; метод оценки — коллективный при обсуждении вариантов реализации элемента бэклога).
2. МежЦОДовая связность (ЦОД — центр обработки данных) (шкала оценки «чем меньше связей между ЦОДами, тем лучше»; метод оценки — подсчет связей между ЦОДами и времени на синхронизацию данных).
3. Отказоустойчивость (шкала оценки — время доступности vs время недоступности сервиса; метод оценки — изме-

рение в «девятках», замер времени на передачу данных и на восстановление контекста при обрыве транзакции).

4. Производительность системы в транзакциях в секунду (TPS) (шкала оценки «чем больше, тем лучше»; метод оценки — нагрузочный тест).
5. Безопасность (шкала оценки — соответствие требованиям ЦБ и НСПК; метод оценки — проверка выполнения требований).

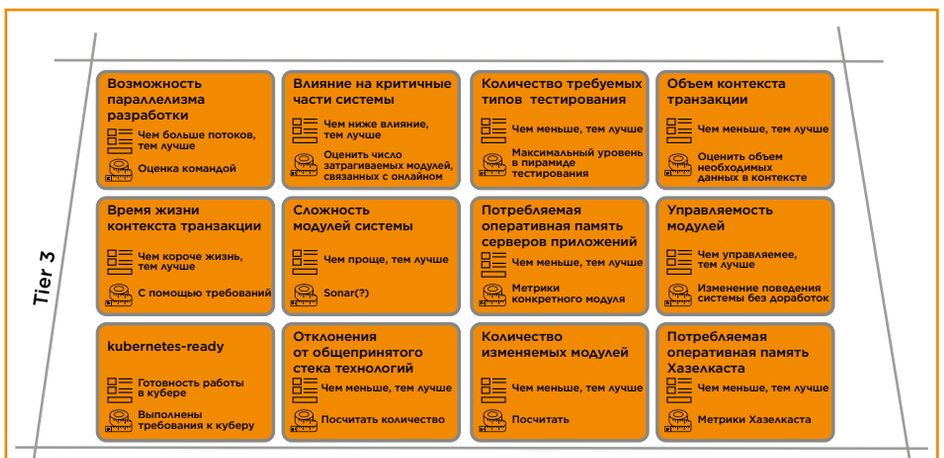
Tier 2. Важные параметры, несоблюдение которых потребует воркэраунд («костыли») для функционирования.

Важные параметры, несоблюдение которых потребует воркэраунд для функционирования

1. Стабильность системы (шкала оценки — количество изменений в системе; метод измерения — подсчет).
2. Утилизация CPU на одну транзакцию (шкала оценки — загрузка CPU; метод измерения — запуск на нагрузочном стенде).
3. Готовность решения к требованиям continuous delivery (шкала оценки «чем меньше отклонение, тем лучше»; метод измерения — deploy в среде continuous delivery (CD).
4. Прямая/обратная совместимость (шкала оценки «совместимость лучше несовместимости»; метод измерения — проверка спецификации).

5. Время на обработку одной транзакции (шкала оценки «чем меньше, тем лучше»; метод измерения — замер на нагрузочном стенде).
6. Горизонтальная масштабируемость (шкала оценки — насколько функция масштабирования ближе к линейной; метод измерения — экспертная оценка).
7. Межмодульная связность (шкала оценки «чем меньше связей, тем лучше»; метод измерения — подсчет связей модулей).
8. Микросервисная архитектура (шкала оценки — совпадение с каноническим определением Фаулера; метод измерения — оценка отклонения).
9. Возможность инкрементального решения (шкала оценки «чем быстрее реализация инкремента, тем лучше»; метод измерения — экспертная оценка).
10. Stateless-реализация (шкала оценки — количество мест, где хранится транзакция во время ее прохождения; метод измерения — проверка спецификации).

Tier 3. Параметры, несоблюдение которых может привести к усложнению разработки и сопровождению системы, возможно, понадобится workaround в обозримом будущем.

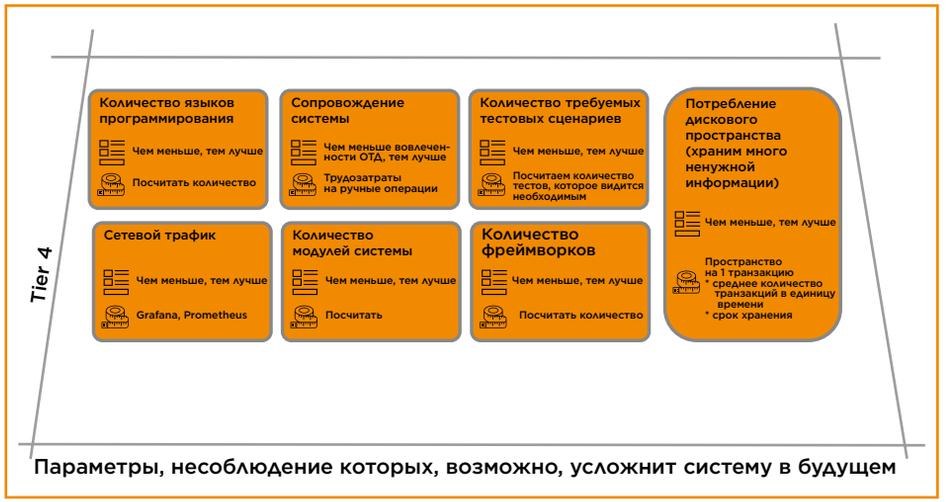


Параметры, несоблюдение которых может привести к усложнению разработки и сопровождению системы, возможно, понадобится workaround в обозримом будущем

1. Возможность распараллеливания разработки (шкала оценки «чем больше потоков, тем лучше»; метод измерения — экспертная оценка).
2. Влияние на критичные части системы (шкала оценки «чем ниже влияние, тем лучше»; метод измерения — посчитать количество изменений в онлайн-модулях).
3. Количество требуемых типов тестирования (шкала оценки «чем меньше разных типов тестов, тем лучше, чем выше по пирамиде тестирования, тем лучше»; метод измерения — подсчет).
4. Объем контекста транзакции (шкала оценки «чем меньше объем, тем лучше»; метод измерения — подсчет необходимых данных в контексте).
5. Время жизни контекста транзакции (шкала оценки «чем короче, тем лучше»; метод измерения — проверка спецификации).
6. Сложность модулей системы (шкала оценки «чем проще, тем лучше»; метод измерения — статический анализатор кода).
7. Потребление оперативной памяти сервера приложений (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет данных, которые будут храниться в оперативной памяти).
8. Потребление оперативной IMDG Hazelcast (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет данных, которые будут храниться в оперативной памяти).
9. Потребление оперативной памяти (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет данных, которые будут храниться в оперативной памяти).
10. Управляемость модулей (шкала оценки — есть управление всеми параметрами или нет; метод измерения — проверка необходимости доработок или конфигурирования).
11. Kubernetes-ready (шкала оценки — готовность модуля работать в кубере; метод измерения — проверка выполнения требований).
12. Отклонение от принятого технологического стека (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет отклонений).
13. Количество изменяемых модулей (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет изменений модулей).

АРХИТЕКТУРНЫЙ ПЛАН ПЛАТЕЖНОЙ СИСТЕМЫ «МИР»

Tier 4. Параметры, несоблюдение которых, возможно, усложнит систему в будущем.



1. Количество языков программирования (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет).
2. Сопровождение системы (шкала оценки «чем меньше операций требуется в процессе сопровождения, тем лучше»; метод измерения — оценка трудозатрат на ручные операции).
3. Количество требуемых тестовых сценариев (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет количества необходимых тестов).
4. Потребление дискового пространства (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет данных, которые будут храниться на дисках).
5. Сетевой трафик (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет данных, которые будут проходить по сети).
6. Количество модулей системы (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет).
7. Количество задействованных фреймворков (шкала оценки «чем меньше, тем лучше»; метод измерения — подсчет).

Метрики в пирамиде оценки могут быть обязательными, может быть придумана композитная оценка в баллах и др. Такой детальный анализ архитектурных решений позволяет выбрать подходящий и оптимальный вариант для решения задачи. Причем на каждом этапе жизненного цикла информационной системы или проекта такая пирамида будет отличаться, т.к. на старте, возможно, приоритет будет отдан критериям, позволяющим сэкономить за счет функциональности или качества проработки программного решения.

За время жизни Мир Plat.Form через прогон архитектурных решений по подобным «пирамидам оценки» сформировался и устоялся определенный технологический стек инструментов, которыми пользуются в компании:

- Java и Kotlin в роли основных языков разработки ПО;
- Liberica JRE 11 — среда выполнения Java-приложений;
- Hazelcast — инструмент распределенного хранения данных в памяти;
- Redis для кеширования горячих данных;
- MySQL для справочных реляционных БД;
- PostgreSQL в роли транзакционного хранилища;
- Kafka для обмена сообщениями и др.

Вы спросите, зачем загонять себя в рамки? Ответ, как ни странно, — кросс-функциональность. Мир Plat.Form под капотом имеет десятки разрабатываемых самостоятельно информационных систем. И если каждая из них начнет жить своей жизнью, то возможности разработки сжимаются до компонентных команд, каждая из которых знает только свой стек технологий.

Конечно же, перечень используемых инструментов регулярно адаптируется под требования современного мира, все новые технологии исследуются коллективным разумом архитектурного сообщества, и если пирамида архитектурных решений дает «добро», то новому инструменту быть.

С КАКИМИ ЗАДАЧАМИ РАБОТАЕТ МИР PLAT.FORM

С архитектурой и ее оценкой разобрались, теперь посмотрим на ее проявления в конкретных задачах. Бизнес финтеха стремительный и неудержимый. Рынок подкидывает задачи как из рога изобилия.

Началось все с карточных транзакций. Самые первые архитектурные решения — стройки ЦОДов для обеспечения функции ОПКЦ (операционно-платежный клиринговый центр системы платежных карт). Тут же потребовались и системы, обеспечивающие маршрутизацию авторизационных сообщений по карточным операциям, расчеты между участниками платежной системы и т.д. Выделился класс mission-critical-систем, от функционирования которых зависит платежная инфраструктура РФ. Для того чтобы системы класса mission-critical бесперебойно функционировали, архитектурные решения в них предусматривают географическое резервирование вычислительной инфраструктуры, каналов передачи данных, распределенное хранение информации, ее дублирование. Также вычислительные ресурсы зарезервированы как минимум в двукратном размере и в любое время все авторизационные сообщения системы платежных карт могут быть обработаны на резервной инфраструктуре.

Как пример, платформа FrontOffice — сердце ОПКЦ, обеспечивающее маршрутизацию платежных сообщений (еще их называют «авторизация») между банками-эквайерами и банками-эмитентами, — основывается на архитектуре типа «одноранговая mesh-сеть». Каждая нода FrontOffice — равнозначный сервер с одинаковым набором программного обеспечения. Роли данного ПО разбиты на шлюзы, маршрутизаторы и кеши данных. Ноды распределены между двумя ЦОДами. Банки-участники подключаются к нодам через прокси прикладных соединений, который знает о состоянии всей сети FrontOffice и умеет переместить состояние сессии в точно работающий сокет точно работающему шлюзу. Так как система обладает требованиями к сверхнадежности, то таких прокси —

несколько инстансов для каждого из ЦОД. А шлюзы также общаются между собой и знают о состоянии каждого — какой из банков на каком шлюзе построил коннект в данный момент. Для распределения топологии используем in-memory data grid Hazelcast и реплицируем конфигурацию на каждую из нод FrontOffice.

Следом появляется идея строительства «платформы лояльности». Новое направление для компании, где важно построить диалог с ритейлом, дав ему инструмент проектирования акций с различными механиками кешбэка для клиентов — владельцев карт «Мир».

Платформа лояльности Privetmir.ru прошла тернистый путь эволюции, который еще не закончен. Ведь это не просто сайт с рекламными объявлениями, а движок расчета кешбэк-акций, механика которых постоянно развивается. И задача по разработке для потребителей этого сервиса (торгово-сервисных предприятий), универсального конструктора подобных механик в совокупности с прогнозом, аналитикой и сегментированием результата акций, нетривиальна. В архитектурном решении есть сложные элементы по обмену данными между защищенным платежным сегментом и зоной интернет-сервисов. Ведь в рамках безопасности данных нам обязательно нужно обеспечить многоуровневую защиту систем, имеющих чувствительные данные, и экстракты, агрегаты из них доставить на сайт — тоже задача не из простых. Для решения такой задачи мы реализовали комплекс программного обеспечения, который позволяет валидировать передаваемые между контурами безопасности данные в режиме plug-and-play, т.е. когда новый валидатор — суть программный код, выполняющий проверку набора данных по любому алгоритму, — можно добавлять в режиме конфигуратора. Здесь используются кластеры Apache Kafka и промежуточное ПО разработки Мир Plat.Form вкупе с межсетевыми экранами.

Также «Мир» вошел в эру мобильных платежей, когда понадобилось обеспечить возможность с карт «Мир» оплачивать товары

и услуги с помощью смартфона. Построенная «платформа мобильных платежей» (ПМП) позволила подключить мобильные кошельки разных производителей. В ней применяются передовые спецификации шифрования, поддерживается множество сценариев аутентификации пользователя и многое другое.

Основной «вызов», решаемый в платформе мобильных платежей, — это то, как быстро можно подключить «кошелек» — мобильное платежное приложение. Если идти путем одного приложения на всевозможные кошельки, то в конце концов этот монолит невозможно будет поддерживать и модифицировать. Архитектура ПМП — это сервисный подход, когда для каждого кошелька строится уникальная часть приложений, управляющих токенизацией. А те компоненты, которые одинаковы для любого из кошельков, переиспользуются.

Запрос с рынка на развитие не позволяет остановиться, и мы разработали и запустили «транспортную процессинговую платформу», которая решает задачу оплаты проезда на транспорте виртуальным билетом, ключом к которому является карта «Мир».

Архитектура транспортного процессинга Bilet.nspk.ru — это объединение многофункционального пользовательского интерфейса (UI) и постпроцессинга данных о поездках с их авторизацией в банках-эквайерах транспортных компаний. Сервисы платежного ядра — это обработчики вокруг топиков Apache Kafka. А пользовательский интерфейс — личные кабинеты пассажира и организаций — это удобные инструменты управления и предоставления информации о поездках, маршрутах, билетах и т.д. Под капотом — Kotlin, Liberica JDK, Kafka, Redis, PostgreSQL, Angular, Camunda, KeyCloak.

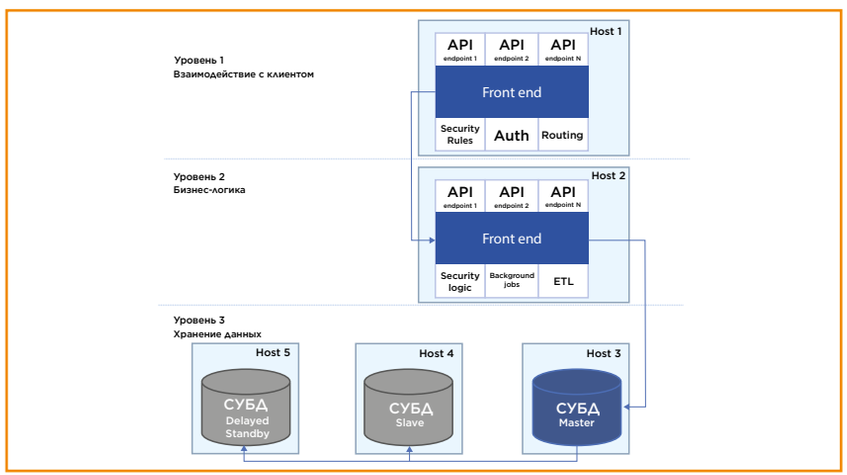
Что стоит за этими бизнес-задачами? Тот самый технологический стек, пирамиды оценки архитектурных решений, в которых, в свою очередь, описаны функциональная, компонентная и интеграционная архитектуры.

Рассмотрим проектирование и выбор архитектурного решения на примере платформы FrontOffice, которая обеспечивает наиболее критичный бизнес-процесс обмена авторизационными сообщениями между банками.

Для оценки были выбраны такие критичные параметры, как:

1. Доступность системы на уровне 99,999.
2. Возможность обновления системы без простоя.
3. Масштабирование системы путем добавления серверов и экземпляров приложений.
4. Модель работы системы на географически распределенных площадках.

Первый вариант архитектуры для оценки — трехзвенная модель.



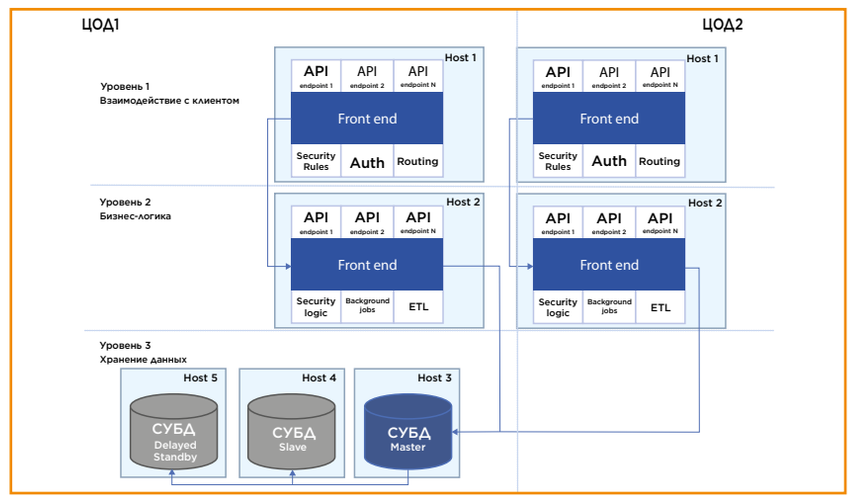
Данный вариант не удовлетворил основным критериям оценки, так как:

1. Не обеспечивал доступность 99,999 — при выходе из строя любого из серверов восстановительные работы заняли бы длительное время, исчисляемое минутами, а то и десятками минут.
2. Не позволял проводить изменения системы без остановки.

3. Отсутствовала возможность горизонтального масштабирования.
4. Не было возможности распределения по площадкам.

Архитектуру требуется доработать.

Следующий вариант учитывает слабые места предыдущего решения, как показано на рисунке ниже:



Данный вариант соответствовал некоторым критериям, но не всем:

1. Не обеспечивал доступность 99,999 — при выходе из строя СУБД восстановительные работы заняли бы длительное время, исчисляемое минутами, а то и десятками минут.
2. Не позволял проводить любые изменения системы без остановки, т.к. обновления СУБД могут требовать остановки.
3. Частично отсутствовала возможность горизонтального масштабирования, т.к. производительность СУБД в данном решении является ограничением всей системы.

Из положительного — появилась возможность распределять некоторые элементы системы на различных площад-

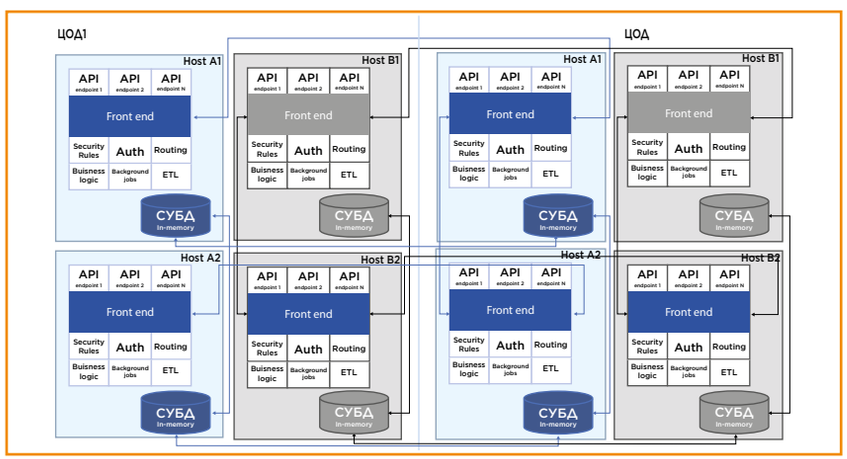
ках, тем самым снижая риск негативных последствий при аварии на одной из них.

Дорабатываем архитектурное решение.

Так как в предыдущем решении центральная роль СУБД не позволяла добиться выполнения основных критичных параметров системы, то было принято решение распределить и СУБД. Мы расположили экземпляры БД в памяти и настроили логику реплицирования данных между как минимум кворумом нод. В итоге получили одноранговую mesh-сеть из одинаковых инстансов FrontOffice.

Для того чтобы была возможность обновлять систему без простоя, половина серверов находится в резервном режиме (hot-standby) и в любое время данные серверы готовы стать активными путем конфигурационных параметров, активируемых без остановки системы. После смены ролей серверов можно обновлять оставшуюся часть системы либо произвести моментальный откат, если что-то пошло не так. Такой режим обновлений еще называют blue-green.

Итоговая архитектура приняла такой вид, как показано на рисунке ниже:



В итоге, пройдя несколько итераций проектирования и оценки, архитектурное решение полностью удовлетворило нефункциональным требованиям, предъявленным к системе. Хотя первый вариант был многим знаком и понятен, критическое мышление и системный подход позволили избежать проблем в дальнейшем.

КАКИЕ КЛЮЧЕВЫЕ АРХИТЕКТУРНЫЕ РЕШЕНИЯ ОБЕСПЕЧИВАЮТ ЗАВЕТНЫЕ 99.999

Для того чтобы оценить архитектурное решение, в пирамиду его оценки нужно добавить некоторые вводные:

1. Насколько важна система для компании (в какой очередности ее восстанавливать, если произошел глобальный сбой).
2. Должна ли система быть высокодоступной для окружающего мира (можно ли отключать систему на длительное время (длительное — это часы и даже дни)).
3. Какое время система может не предоставлять свои сервисы клиентам за период времени (RTO — recovery time objective).
4. За какое время до сбоя система может потерять данные (RPO — recovery point objective).

Третий пункт обычно влияет на такую характеристику, как SLA — service level agreement, что зачастую компания обещает клиентам или акционерам.

Доступность часто измеряется в «девятках». Единица — это когда система во всем периоде непрерывно предоставляет свой сервис. Например, сайт, который никогда не выдаст ошибку 500 (internal server error). Если доступность определяется как «пять девяток», или 0,99999, это значит, что за определенный период (например, день, месяц, год) система может не предоставлять клиентам свои сервисы в течение 0,00001 от этого определенного периода. Например, если целевая доступность «пять девяток в год», то это значит, что

сервис может потенциально быть недоступен 5 минут и 15 секунд за год (или 5 минут и 16 секунд, если год високосный). Это только пример, когда отчетный период для расчета SLA равен календарному году. Но здесь можно формулировать различные схемы расчетов, зависящие от целей. Например, можно измерять доступность сервисов системы «каждые прошедшие 30 дней», «ежедневно» и т.п.

Важно на этапе разработки учесть требование по измеряемости сервиса. Это могут быть как механизмы внутри компонентов системы, которые точно знают, что система в момент времени недоступна, и предоставляют информацию о времени начала недоступности и о времени восстановления. Это могут быть специальные «пробники», находящиеся «за периметром», эмулирующие действия пользователя, а также логирующие все инциденты, связанные с недоступностью сервиса.

Такие метрики служат как средством контроля SLA, так и помогают развивать архитектуру системы, выявляя слабые места, тем самым позволяя устранять те или иные недостатки, приводящие к снижению качества сервиса.

Но что, кроме доступности, должно учитываться разработчиком высокодоступной, надежной распределенной информационной системы? Под разработчиком будем понимать не только того, кто пишет программный код, но и любого участника продуктовой команды. Как в управлении проектом есть ВДК (время, деньги, качество), так и в распределенной системе есть троица — CAP (Consistancy — консистентность, целостность данных; Availability — доступность, о чем мы говорили выше; Partitioning resistance — готовность к «расщеплению» данных, когда, допустим, в одном ЦОД есть данные по одному клиенту, а во втором ЦОД этих данных нет).

CAP-теорема гласит, что в распределенном приложении возможно достижение только двух показателей.

CA. Система со всегда целостными данными и всегда доступна. В данном сценарии распределенная система должна состоять из набора идентичных экземпляров, полностью реализующих всю логику сервисов и хранение данных. Каждый экземпляр может обработать только «свои» данные. Таким образом, система доступна, пока работает хотя бы один ее экземпляр, и данные, относящиеся к этому экземпляру, всегда консистентны. Но как только мы обратимся с данными к «чужому» экземпляру — получим ошибку. Такой подход называется «шардинг» (от англ. shard — часть, фрагмент), когда один экземпляр показывает, например, покупки клиентов, фамилии которых начинаются с буквы А, второй — с Б и т.д.

AP. Система всегда доступна, и ей нестрашно разделение данных. Вариант, который может называться еще как eventually consistent — иногда или с прошествием времени консистентный. Такой вариант работы распределенной системы позволяет обеспечить доступность системы путем распределения ее экземпляров, при этом каждый из экземпляров обрабатывает запросы без разделения. Но в ответ на запрос система необязательно вернет актуальную информацию. Например, два экземпляра системы работают одинаково, показывая покупки клиентов за последний час. Но в одном экземпляре неполный их перечень — не хватает самой свежей покупки, потому что еще не завершилась синхронизация между хранилищами первого и второго экземпляра.

CP. Система со всегда целостными данными, и ей нестрашно разделение данных. Вариант распределенной системы, когда хранилище данных общее. То есть все экземпляры системы, обрабатывающие запросы пользователей, обращаются за информацией к единому компоненту, являющемуся одной «точкой истины». В таком случае при отсутствии связи между каким-либо экземпляром фронтенда и хранилищем система не сможет обработать запрос пользователя.

На практике данные режимы приходится совмещать, чтобы свести к минимуму риски потери консистентности, недоступности или неполноты данных в ответах на запросы пользователей системы. Например, в систему, работающую в режиме CP, можно (и нужно) добавлять локальный кеш к каждому ее

экземпляру, тем самым снижая зависимость от центральной базы данных.

В сервисы, реализованные по AP-режиму, можно добавить арбитра, который отслеживает состояние репликаций и перенаправляет запросы в более полную реплику. Этакий умный балансировщик (и такие есть в Мир Plat.Form).

> Умный балансировщик — так называется подход, при котором мы не только мониторим состояние серверов и их данных, на которые будем перенаправлять запросы от клиента, но и когда мы прорабатываем ключевые данные в протоколе уровня представления, на основе которых можно приоритетно дать задание наиболее подходящему обработчику данных. Например, в поле instance_id мы записываем в первом из цепочки сообщений имя сервера, на котором оно обработалось, а в имени сервера кодируем и номер ЦОД, в котором этот сервер находится, и «прокидываем» это значение в каждом сообщении-потомке. При поступлении следующего сообщения, относящегося к той же транзакции, балансировщик отдает приоритет серверу, указанному в instance_id, т.к. при необходимости восстановления контекста цепочки сообщений сервер, который занимался предыдущими сообщениями, справится с этим лучше всех остальных.

ЗАКЛЮЧЕНИЕ

Как вы уже поняли, в мире информационных технологий нет стандартного рецепта изготовления информационных систем и платформ, предоставляющих сервисы. Но формирование подходов к реализации однотипных задач, таких как моделирование и хранение информации, шардинг данных, умная балансировка запросов, предоставление метрик доступности сервиса и другие архитектурные решения позволяют разрабатывать системы с заданными характеристиками и функциональностью. При этом выбор инструмента вполне разумно ограничить теми вариантами, которые прошли апробацию сообществом как вне компании, так и внутри.

Архитектура информационных систем — это прежде всего процесс. Процесс постоянного улучшения, оптимизации, процесс, который адаптируется под текущие и будущие вызовы рынка. И уже во вторую очередь информационная архитектура — это артефакты в виде схем, перечней функциональности, описания требуемого оборудования и т.д. В IT архитектурой занимается каждый, перманентно изменяя конфигурационные параметры, дописывая код приложения, тестируя или заводя дефекты.

FRONTOFFICE: КАК ПОДДЕРЖИВАТЬ ПЯТЬ ДЕВЯТОК И НЕ УТОНУТЬ В LEGACY

Пойдем вглубь и в детали. Что происходит, когда вы прикладываете карту к терминалу? Как обрабатывается платеж? Куда он попадает? Как обеспечить тысячи транзакций в секунду? И во что они выливаются внутри системы?

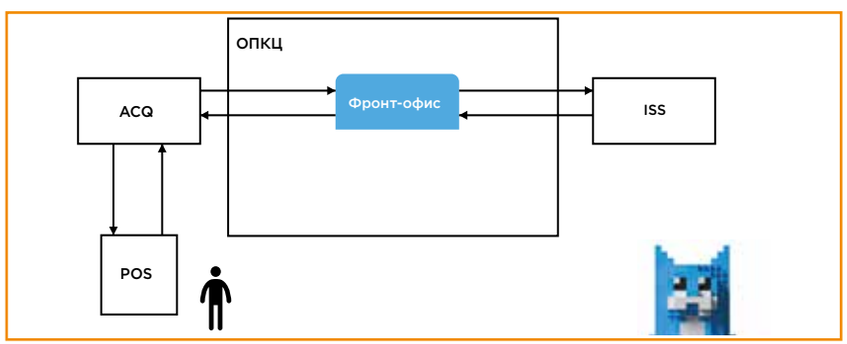
Об этом мы поговорим во второй главе про устройство самого тяжелого компонента платежной системы — FrontOffice.

Мы сделаем все правильно. Сначала рассмотрим требования, которые предъявляются к FrontOffice: устойчивость системы, резервирование, гарантии доступности, производительность и доставка сообщений. Затем поймем, как эти требования реализуются на уровне архитектуры.

Узнаем все из первых рук — от руководителя группы разработки компании Мир Plat.Form Юрия Бабака.

ЧТО ТАКОЕ FRONTOFFICE

Тут можно было бы поговорить про JS, Angular, React и другие модные фреймворки, но мы начнем с бэкенда. Для начала разберемся, что же такое FrontOffice.



Мир Plat.Form использует POS-терминалы (Points of Service). Это могут быть, например, терминалы в магазине, к которым вы прикладываете карту. Эти устройства выдает банк-эквайер, а вашу банковскую карту выпускает банк-эмитент. FrontOffice — это система онлайн-обработки финансовых сообщений между банком-эквайером и банком-эмитентом.

Из FrontOffice различных платежных систем и клиринга состоит операционно-платежный клиринговый центр (ОПКЦ). Это одна из тех больших систем, которые Мир Plat.Form предоставляет банкам-партнерам в качестве сервиса. У этой системы тяжелый трафик и множество одновременных финансовых сообщений.

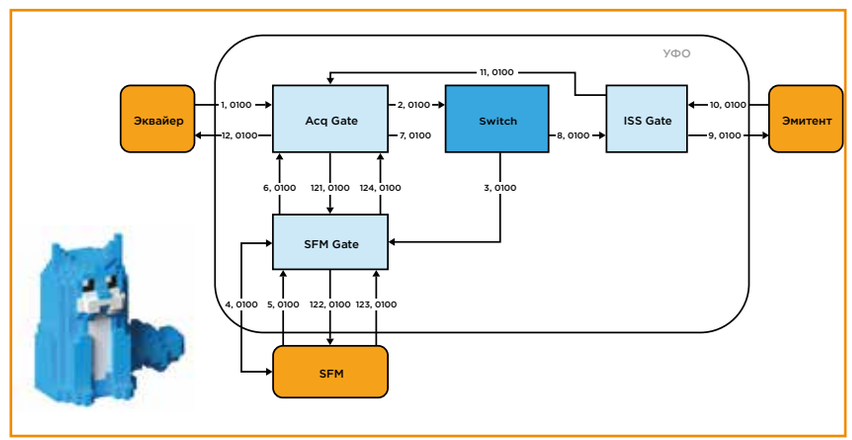
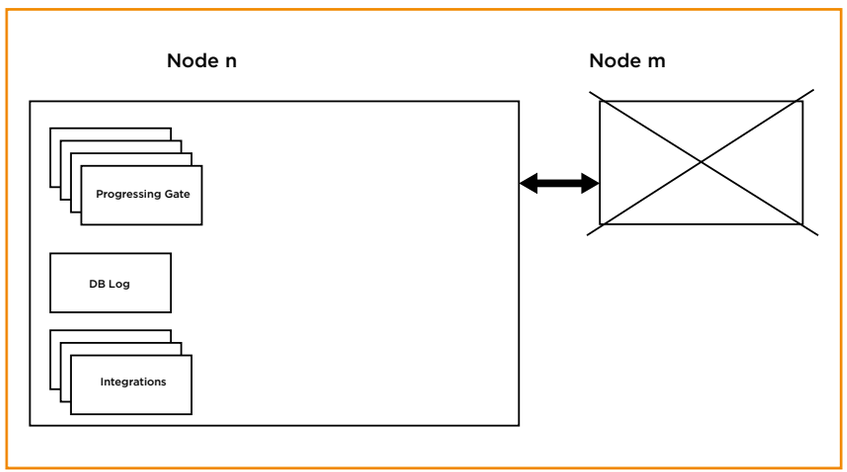


Схема Happy Path — самый примитивный случай отправки запроса на списание денежных средств, так называемое сотое сообщение с кодом 0100. Чтобы от банка-эквайера пришло сообщение с результатом, нужно проделать минимум 12 операций. В системе обеспечивается прохождение тысяч TPS — это минимум 12 тысяч операций. Разумеется, они должны быть атомарными, что создает дополнительные сложности. Когда у финансового сообщения еще более непростой жизненный цикл, может быть и 20 тысяч операций. Конечно, к системе, которая обеспечивает обработку такого трафика, достаточно жесткие требования.

ТРЕБОВАНИЯ К FRONTOFFICE

Чтобы спроектировать подобный FrontOffice самостоятельно, нам необходимо реализовать в системе следующие требования.

УСТОЙЧИВОСТЬ СИСТЕМЫ К ЛЮБОМУ ИЗМЕНЕНИЮ



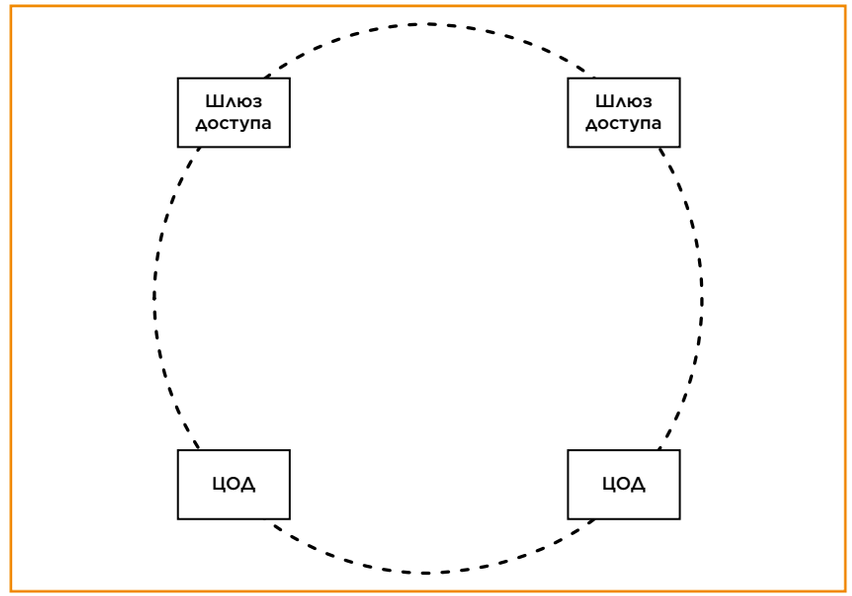
В распределенной системе с кластером и узлами необходима устойчивость к изменению топологии этого кластера во время обработки сообщений, и нужно быть готовыми к проблемам с узлами. Например, может отказать процессинговый сервис. Тогда другие должны взять его работу и сбалансировать нагрузку.

РЕЗЕРВИРОВАНИЕ

Это требование достаточно очевидно — нужно резервировать все, что только возможно:

- шлюзы доступа;
- ЦОДы;
- кластеры;
- сервисы внутри этих кластеров.

FRONTOFFICE: КАК ПОДДЕРЖИВАТЬ
ПЯТЬ ДЕВЯТОК И НЕ УТОНУТЬ В LEGACY



Почему надо резервировать шлюзы доступа? Потому что случаются технические повреждения линий связи. Чтобы к ним подготовиться, нужно несколько сетевых маршрутов.

ГАРАНТИИ

Необходимо обеспечить следующие гарантии:

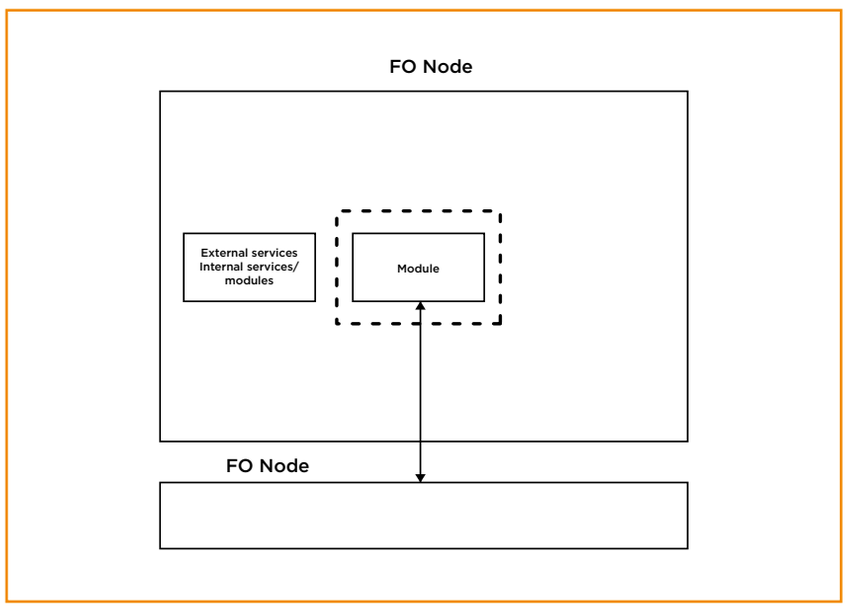
1. **Доступность** на уровне пяти девяток — финансовое сообщение должно дойти в любое время и при любой ситуации.
2. **Производительность** — Мир Plat.Form не может позволить себе долго держать финансовое сообщение в своем контуре, транзакция должна быть обработана максимально быстро. На всю цепочку: POS-терминал, банк-эквайер, процессинг и банк-эмитент — должно уходить 2-3 секунды. Чуть больше, если есть какие-то проблемы в сети. Вспомните, сколько времени у вас занимает банковская операция в магазине.
3. **Доставка сообщений** — не должно быть ситуаций, когда при проведении финансового сообщения банк-эмитент

временно блокирует средства или списывает их без уведомления банка-эквайера. Как не должно быть и обратной ситуации, при которой придется искать концы и все равно списывать с кого-то деньги.

Следует учитывать, что устойчивость к изменениям нужна не только платформе в целом, но и всем ее компонентам.

УСТОЙЧИВОСТЬ КОМПОНЕНТОВ

Каждый компонент должен быть готов штатно завершить работу в полной изоляции, что накладывает дополнительные требования.

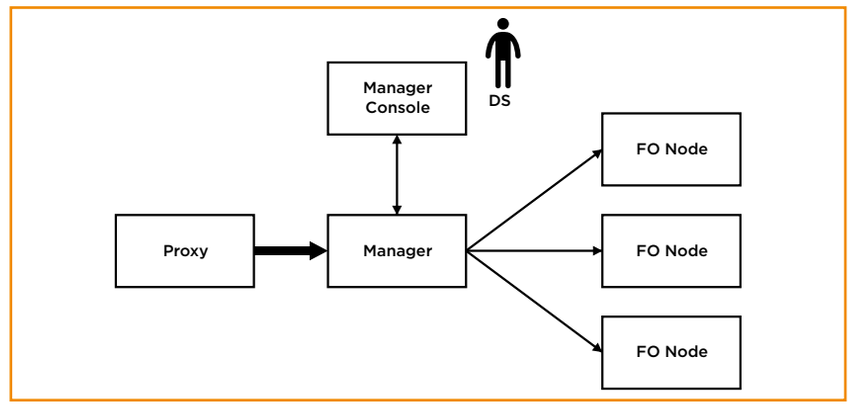


На схеме показано, что, если процессинговый узел понимает, что остался один, он должен завершить работу и записать, что происходило и какие транзакции сейчас в обработке. Это штатный режим работы, и он должен быть обеспечен при любых обстоятельствах.

ПРИКЛАДНАЯ АРХИТЕКТУРА

Чтобы реализовать перечисленные требования, в Мир Plat. Form используется распределенная система — кластер. Все узлы в нем знают о других и имеют с ними связь. Также система содержит самописный балансировщик, состоящий из трех частей:

1. Proxy — проксирует трафик и переводит на Manager.
2. Manager — перераспределяет трафик на ноды.
3. Console управления для менеджера.



Это небольшое веб-приложение, при помощи которого сотрудники дежурной службы 24x7 осуществляют мониторинг за продуктовыми кластерами. Управлять нагрузкой на кластер можно, переводя трафик с одного узла на другой. Это происходит, например, если возникает потребность в увеличении пропускной способности.

ИДЕМ В ДЕТАЛИ

Немного подробностей об устройстве самого кластера. С кластером все понятно — это несколько машин, соединенных сетью. Давайте поговорим о тех сервисах, которые реализованы внутри каждого узла и обеспечивают его работоспособность.

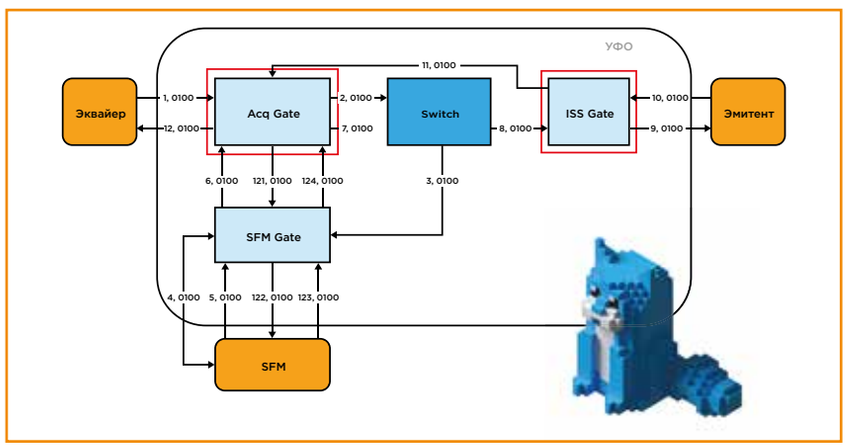
МЕНЕДЖЕР УЗЛА

Node Manager, грубо говоря, точка входа в узел, «запускатор», но у него есть и другие зоны ответственности. Он не только запускает и останавливает модули и службы внутри каждого конкретного узла. Он знает о статусах всех компонентов внутри каждого узла и может публиковать их, чтобы было видно состояние всего кластера. Также Node Manager отвечает за синхронизацию распределенного состояния.

УЗЛЫ ОБРАБОТКИ

Можно сказать, что самыми главными являются узлы обработки, которые:

- содержат логику обработки сообщения;
- занимаются журналированием сообщений и их обработкой;
- являются точкой входа сообщений в кластер / выхода сообщений из кластера, к которой могут подключаться эквайеры и эмитенты.

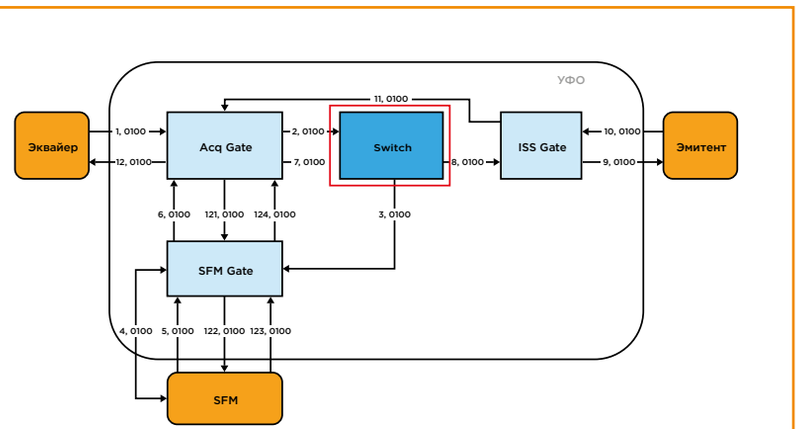


После того как узлы обработки выполнили свою часть бизнес-логики, они в том месте жизненного цикла, где это необходимо, пересылают сообщения на маршрутизатор.

МАРШРУТИЗАТОР

Он владеет всей информацией о текущей топологии кластера. В задачи маршрутизатора входят:

- построение маршрутов;
- передача сообщений в кластере;
- генерация ID операций в платежной системе.

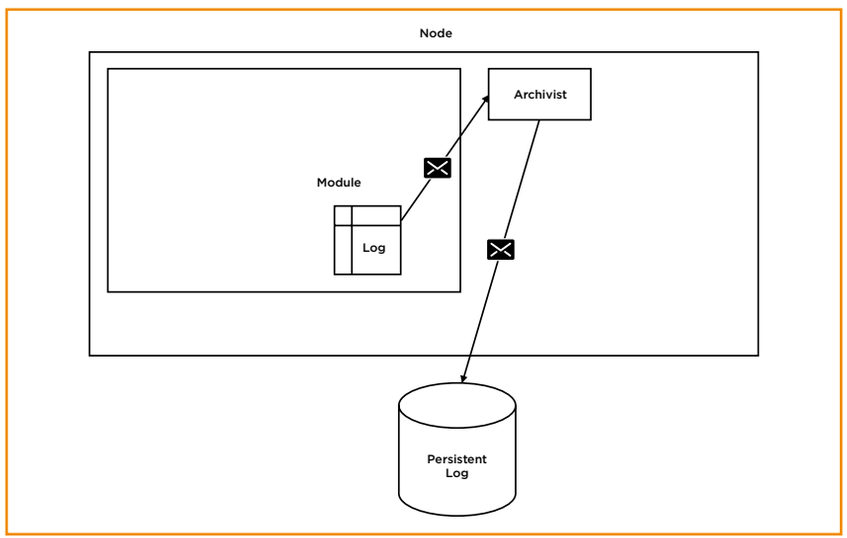


На схеме показаны эквайер и эмитент. Эквайер подключается к шлюзу и присылает финансовое сообщение. Его обрабатывает процессинговый узел, но не передает напрямую в эмитент, потому что каждый узел обработки в Мир Plat.Form полностью stateless, а отправляет сообщение на локальный маршрутизатор внутри каждого узла. Маршрутизатор знает, где находится целевая нода для построения маршрута финансового сообщения. При необходимости он может строить бэкап-маршруты. Он знает о текущем состоянии и может перенаправить сообщение, если топология изменится.

После этого происходит передача сообщения в кластере. Для всех финансовых операций генерируются ID, которые должны быть уникальны для всего кластера в рамках определенного временного периода.

АРХИВ

Вся информация об обработанных сообщениях находится в персистентном хранилище (СУБД, «холодное» хранилище). Эти данные необходимы для аналитики и клиринга. Это выглядит так, как показано на рисунке ниже:

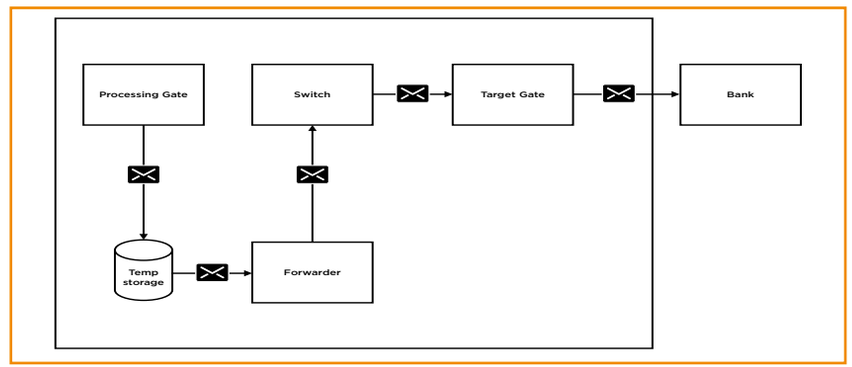


Исходя из требований, например, готовности работать в полной изоляции, мы не можем открывать прямое подключение к базе. Для этого есть локальный буфер, где хранятся логи с файловой системой каждого конкретного узла. Этот сервис ходит по модулям, забирает логи и складывает их в хранилище, если оно доступно. В случае, когда доступа в хранилище нет, сервис выполняет определенный алгоритм действий.

STORE-AND-FORWARD

Требования по гарантированной доставке реализуются сервисом отложенной доставки Store-and-Forward, который выглядит так, как показано на рисунке:

FRONTOFFICE: КАК ПОДДЕРЖИВАТЬ
ПЯТЬ ДЕВЯТОК И НЕ УТОНУТЬ В LEGACY



Если на текущем шлюзе обработки адресат недоступен, сообщения складываются во временное персистентное хранилище, из которого их забирает специальный сервис Forwarder. Он выступает в роли виртуального обработчика и возвращает сообщения в стандартный жизненный цикл обработки. Это нужно для того, чтобы на узлах обработки не скапливался state.

Эти сообщения должны храниться в персистентном хранилище, потому что адресат может быть недоступен длительное время. Причем сразу отправить адресату все, что накопилось, нельзя. Надо учитывать, что у адресата, например, может быть проблема с производительностью и при отправке ему 10 тысяч финансовых сообщений в одном пакете, мы его завалим. Для этого в Forwarder реализована регулировка пропускной способности сервиса.

ИНТЕГРАЦИИ

Помимо локальных сервисов, которые есть на каждом узле, в Мир Plat.Form есть интеграция с 35 реализованными системами. Вот основные:

1. Master Data: справочники, информация о банках и карточках, актуальные курсы валют.
2. Back Office для офлайн-обработки финансовых сообщений, которые отправляются в Центробанк.
3. Резервная авторизация позволяет одобрять или не одо-

брять финансовые операции при недоступности банка-эмитента, если этот банк дал такое право.
4. Фрод-мониторинг и риск-менеджмент.
5. Токенизация: сервисы поддержки работы кошельков Mir Pay, Samsung Pay, Apple Pay, которые превращают токен, записанный в кошелек на телефоне в номер карты, и возвращают жизненный цикл финансовой операции на понятный путь.
6. ИППС (Интеграционная Платформа Платежных Систем) позволяет интегрироваться с партнерами из ближнего зарубежья и обслуживать банковские карты друг друга за рубежом.

КАК МЫ ГОТОВИМ HAZELCAST

В качестве DataGrid Мир Plat.Form использует Hazelcast. Он применяется для агрегации текущего актуального состояния кластера. В Hazelcast каждый Note Manager публикует не только свое состояние, но и состояние всего локального узла, поэтому всегда известно полное текущее состояние кластера. Для мониторинга есть REST API.

Также через Hazelcast обеспечивается работоспособность операций CMC (Single Message).

Если есть вероятность потери транзакций, которые находятся в работе, Hazelcast позволяет переносить текущие контексты на другой узел по команде оператора менеджера соединений.

По архитектуре все, но еще стоит остановиться на качестве и скорости.

БОЛЬШЕ ТЕСТОВ БОГУ ТЕСТОВ

Мир Plat.Form использует следующие виды тестов:

- юнит-тесты;
- интеграционный регресс в рамках одной системы, когда поднимается весь FrontOffice и проходят end-to-end-тесты;

- интеграционное тестирование между различными системами, когда поднимается FrontOffice, BackOffice и балансировщик;
- нагрузочное тестирование;
- стресс-тестирование для проверки пиковых нагрузок. Есть локальные пики, которые мапятся, например, во время ланча или пятничных покупок, и более экстремальные, такие как 31 декабря, когда все покупают подарки перед Новым годом;
- исследовательское тестирование для понимания, как изменение платформы повлияет на производительность перед реальными изменениями;
- тестирование тестирования для проверки корректности самих тестов.

Тестирование позволяет в Мир Plat.Form сохранить хороший Time to Market (TTM). Поставка в продакшен происходит каждые два спринта. Спринты двухнедельные, поэтому TTM получается в районе 4 недель. Объем каждой такой поставки около 8 достаточно объемных бизнес-фич. Например, поддержка кешбэка по Ростуризму или выдача наличных в кассе. Помимо поставки бизнес-фич, происходит работа с техдолгом, который берется у каждого спринта. А еще сложные рефакторинги на платформе: выделение новых сервисов или модулей или изменение алгоритмов работы платформы.

ЗАКЛЮЧЕНИЕ

При обеспечении максимальной надежности и максимальной доступности (это обязательное требование к любой платежной системе) Мир Plat.Form сохраняет максимальную скорость разработки. При необходимости можно сократить TTM до 1-2 недель, но планирование обычно позволяет этого избежать.

ТЕСТИРОВАНИЕ ОТКАЗОУСТОЙЧИВОСТИ ВЫСОКОНАГРУЖЕННЫХ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Пойдем еще глубже. В прошлых главах мы рассмотрели верхнеуровневую архитектуру, сформированную исходя из требований, которые предъявляются к платежной системе. Ключевое из них — отказоустойчивость. Минута простоя стоит очень дорого.

Как достигается отказоустойчивость, мы уже знаем, рассмотрим, как она тестируется. Сначала поговорим о потенциальных кейсах: недоступность ЦОД, сетевые зависания, нехватка ресурсов, ошибки кода и архитектуры, человеческий фактор. А затем опишем стратегию и модель тестирования.

В этом нам поможет ведущий разработчик по автоматизации тестирования Мир Plat.Form Дмитрий Цитман.

МЕТРИКИ И НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Нефункциональное тестирование построено на бизнес-требованиях к самой системе (например, определенный TPS) и тестировании ресурсов.

В Мир Plat.Form требования разделены на несколько категорий:

- **данные**, завязанные на тестировании отказоустойчивости, — например, процент поврежденных данных после возникновения кейса отказа, когда система повела себя неопределенно;
- **функционал** — например, метрики производительности, время отклика и доступность функционала;
- **приложения**: garbage collector, утилизация хипа, работа потоков и все, что с этим связано;
- **ресурсы**: утилизация CPU, потребление оперативной памяти, сетевого канала, количества и коннектов.

В рамках тестирования лучше всего собирать метрики, связанные с алертами, настроенными на промышленной среде. Тогда при возникновении отказа можно оценить, локализуется ли этот кейс с помощью системы мониторинга или нет.

ПОТЕНЦИАЛЬНЫЕ КЕЙСЫ ОТКАЗОВ

Рассмотрим кейсы по каждой группе дефектов отказоустойчивости. Всего их 6: недоступность ЦОДа, системные зависания, нехватка ресурсов, ошибки кода и архитектуры, а также человеческий фактор. В каждом примере система не соответствует требованиям или демонстрирует нерегламентированное поведение. Это нужно, чтобы разобраться, на чем фокусироваться для решения этих кейсов.

НЕДОСТУПНОСТЬ ЦОДА

Это глобальный тип отказов, связанный с **недоступностью в георазпределенной системе**. Причиной может быть потеря связи между центрами обработки данных или повреждение одного из ЦОДов. Например, отключилось питание и были потеряны уникальные только для него данные. Тогда на одном ЦОДе можно провести те операции, которые были на другом, — получить незапланированные из-за его недоступности ситуации и прояснить проблему рассинхронизации данных при использовании нескольких активных ЦОДов. Она чаще всего в модулях, связанных с взаимодействием между ЦОДами: репликация и редактирование данных на одном ЦОДе или сбор информации с других.

СЕТЕВЫЕ ЗАВИСАНИЯ

Причинами зависаний могут стать: **баги сетевого оборудования, нехватка канала или проблемы среды виртуализации** (например, лаги виртуальной среды, если система развернута в VMware).

Чтобы найти причины, нужно оценивать два типа модулей. Во-первых, элементы с большим числом сетевых обменов и ответственные за передачу больших объемов данных или

больших файлов. Во-вторых, элементы с высоким объемом трафика, то есть с высокой частотой обменов, например модули, работающие с кешем, а особенно модули, которые обмениваются с кешем много раз за транзакцию. Если таких вызовов на каждую транзакцию десятки, то отклик кеша может увеличиться с 200-300 микросекунд до нескольких десятков или даже сотен миллисекунд. Это, в свою очередь, может увеличить время на транзакцию до 4-6 секунд. После этого последуют другие проблемы внутри системы.

НЕХВАТКА РЕСУРСОВ

Этот тип отказов связан с использованием сетевых ресурсов: CPU, RAM либо таких конфигурационных ресурсов, как количество сокетов соединений или дескрипторов обмена данными на ОС. Отказ может случиться **при неправильной оценке ресурсов**, если провели некорректное нагрузочное тестирование, неверную оценку потребления системы или не рассчитали дополнительную нагрузку на событие. Например, не учли «черную пятницу». Если в этот момент возникнет другой отказ, который приведет к нехватке ресурсов, то может получиться эффект домино. Тогда будет еще и высокая стоимость по ресурсам других инцидентов. Поэтому особое внимание следует уделить модулям, которые больше других потребляют мощности системы: те, что связаны с онлайном либо с критическим функционалом и/или тяжелыми расчетами. Оценка мощностей подскажет наиболее уязвимые места.

ОШИБКИ КОДА

Ошибки кода чаще всего возникают из-за того, что тестами невозможно покрыть какие-то части системы, либо из-за просчетов при ее разработке. Причиной отказа может стать **незапланированная блокировка или некорректная работа с потоковыми менеджерами**. Например, вы начали работать с одним и тем же менеджером потоков сразу несколькими частями системы. Разумеется, неважная часть просто заглушит более важную, и система начнет заваливаться.

Также это могут быть специфичные утечки, когда при написании кода не учитывались уникальные ситуации, а из-за

высокой нагрузки на систему в промышленной среде их невозможно воспроизвести на тесте юнит-уровня или в интеграционном тестировании. Такие ошибки могут возникать в любом модуле или даже в нескольких, поэтому необходимо выбрать критически важный функционал в системе.

ОШИБКИ АРХИТЕКТУРЫ

Данный тип отказов близок к предыдущему, но связан с **неправильным проектированием системы и/или неверными расчетами**, когда возникает незапланированное действие, мало отличающееся от штатного, но приводящее к отказам. Например, при попытке проведения транзакций происходит рассогласование данных как во внутренней системе, так и во внешних носителях. В этом случае рассматриваем не отдельные модули, а архитектурные проблемы, связанные с критичным функционалом и высоким потреблением ресурсов. Это может проявиться для большого количества модулей, необходимых для горизонтальной масштабируемости системы или отвечающих за безопасность системы в случае отказа того или иного модуля.

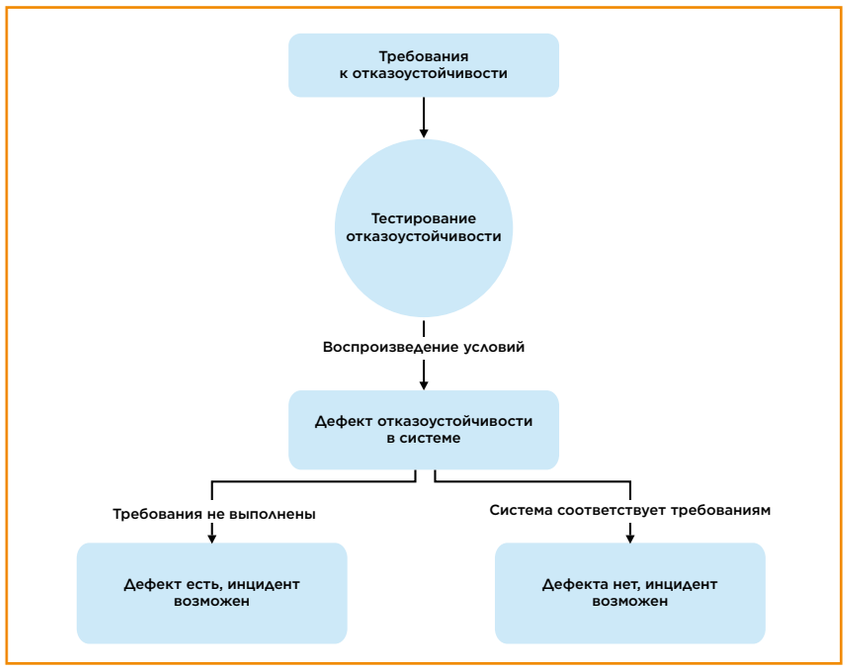
ЧЕЛОВЕЧЕСКИЙ ФАКТОР

Эксплуатация системы сопровождается массой различных работ, и порой при перезагрузке части системы происходят ошибки, связанные с подсистемами временного хранения. Конечно, предполагается, что кеш в кластере, а потому защищен, но бывают действительно грубые ошибки, которые приводят к серьезным отказам в системе. Помимо этого, рассинхрон может случиться из-за проведения параллельных работ, если система изначально не рассчитана на такую активность. Обращаем внимание на **модули, связанные с кешем или с блокирующим функционалом**, как наиболее важным.

ОТКАЗОУСТОЙЧИВОСТЬ

Если система умеет противостоять описанным выше случаям, она отказоустойчива и соответствует нефункциональным требованиям. Поэтому собираем их при тестировании и проверяем, происходит ли отказ. Триггером становятся

изменения в системе, которые способны ее разрушить. Они могут не проявиться либо воспроизвестись, но не нарушить требований, — тогда проблемы в системе нет.



ПРИОРИТИЗАЦИЯ ДЕФЕКТОВ ОТКАЗОУСТОЙЧИВОСТИ

Существует всего 6 групп дефектов отказоустойчивости, но из них необходимо выбрать наиболее важный, чтобы протестировать его в первую очередь. Если тестировать все подряд, то можно пропустить действительно серьезные проблемы. Для этого в Мир Plat.Form расставили приоритеты по масштабу, вероятности и последствиям от дефектов отказоустойчивости.

Масштаб

Речь о местах в системе, где уязвимость начинается и приводит к отказу. Например, внешняя система не ответила на запрос, что привело к каким-то последствиям. Повлиять на внешнюю систему нельзя, поэтому это наиболее опасная группа по масштабу возникновения проблем.

После этого идет глобальный отказ внутри системы. Например, нескольких модулей или подсистем, связанных с одним и тем же дефектом.

Третий уровень, когда дефект или уязвимость связаны с одним конкретным модулем внутри системы, то есть он внутренний локальный.

Вероятность

Здесь тоже три группы: сначала отказы, которые происходят регулярно с высокой частотой — до нескольких раз в месяц. Но даже один отказ в месяц — это серьезная проблема, особенно если у него есть последствия.

После этого — отказы, происходящие редко, до нескольких раз в год. Они могут быть связаны с какими-то событиями, мероприятиями или праздниками.

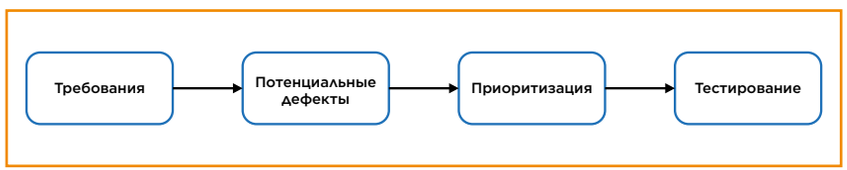
Еще есть абстрактные или уникальные отказы. Они происходят только при возникновении сразу нескольких триггеров внутри системы, но возникают редко и еще реже повторяются.

Последствия

Следующий по приоритету дефект отказоустойчивости — последствия. У него есть масштаб влияния. Например, если отказал только тот модуль, в котором находится дефект, а последствия возникшего отказа можно исправить внутри системы — это **поправимое последствие**. А если дефект существует в одном модуле, а отказ распространился на всю систему, то масштаб влияния будет другим. Бывают ситуации, когда повреждены данные, которые согласуются сразу с несколькими элементами системы. Тогда восстановить их, особенно если система под высокой нагрузкой, крайне сложно. При этом время недоступности критического функционала влияет на репутацию. Это нарушение гарантий по доступу к критически важному функционалу, на который завязано SLA. Такие последствия называют **непоправимыми**.

К последствиям также относят трудозатраты на восстановление системы после отказа. На устранение одних инцидентов уходит немного времени, а для решения других нужно сразу несколько уникальных специалистов. Например, в сезон отпусков их отсутствие может привести к невозможности восстановить отказ в кратчайшие сроки, что равно серьезной проблеме внутри системы.

После распределения приоритетов можно переходить к тестированию на отказоустойчивость.



ТЕСТИРОВАНИЕ

СТРАТЕГИЯ ТЕСТИРОВАНИЯ

Определяем часть системы, в которую входит наибольшее количество модулей, связанных с отказом. За счет ограничения скоупа тестирования повышается его качество. Чем больше проверяемая часть, тем больше шансов получить невалидные результаты или другой отказ.

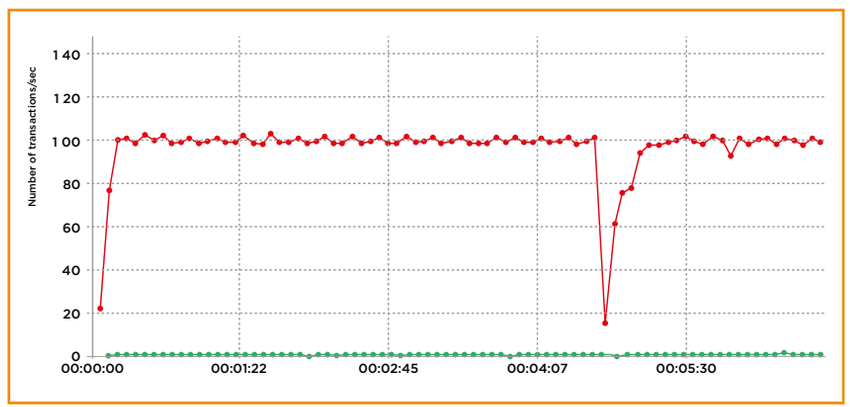
После ограничения скоупа тестирования выставляем нефункциональные требования к системе. Они помогут четко разделить ситуации, когда система соответствует требованиям, а дефект, с точки зрения бизнеса, не представляет опасности для критического функционала или репутации. Также нужны критерии успешности тестирования, чтобы точно воспроизвести нужный дефект и получить ожидаемые результаты. Проверка осуществляется соответственно проверяемому дефекту. После этого переходим к модели тестирования.

МОДЕЛЬ ТЕСТИРОВАНИЯ

Инженеры Мир Plat.Form используют любые необходимые инструменты: LoadRunner, JMeter, Gatling или самописные генераторы нагрузки. Поскольку скоуп тестирования ограничен, применяются заглушки, которые не должны аффектить систему во время инцидента.

Сценарий нагрузочного тестирования отказоустойчивости — это обычный load-тест. Стресс-тестирования не проводятся. Главная цель — определить, как поведет себя система при возникновении триггера отказа при нормальной эксплуатации, а не выяснить, как она ведет себя на бешеной нагрузке.

После воспроизведения инцидента лучше подождать и посмотреть, восстановится система или нет. Например, на графике ниже видно, что после отказа система смогла это сделать. И тогда остается выяснить, как долго функционал не соответствовал уровню нагрузки.



Поэтому минимизируем профили. Модель может быть очень сложной, а при ее упрощении повышается качественное тестирование. Расширять скоуп имеет смысл только в случае, если в системе присутствует критически важный функционал. Даже не связанный с появлением отказа, он позволит оценить влияние дефекта на функционал.

АВТОМАТИЗАЦИЯ

Важная часть тестирования — восстановление системы после проверки отказов. На восстановление из деплоя ухо-

дит слишком много времени, поэтому в Мир Plat.Form автоматизировали цикл всего тестирования:

- деплой системы;
- тестирование;
- анализ;
- вывод;
- повторное тестирование в случае необходимости.

При этом остаются проблемы, которые сложно предугадать.

ПОДВОДНЫЕ КАМНИ

Трудно воспроизвести ситуацию для случайного события, то есть которое воспроизводится не каждый раз. Чтобы снизить хаотичность тестирования, повысить воспроизводимость результатов и удешевить процесс, в Мир Plat.Form упрощают модель тестирования (ограничивают скоуп) и ужесточают критерии его проведения, то есть требования к его успешности.

Метрики при тестировании отказоустойчивости в распределенных системах менее стабильны, чем при тестировании последовательных систем, что также является проблемой. Для ее локализации в Мир Plat.Form используют композитные метрики, например сумму утилизации CPU инстансов определенного типа сервисов — она более устойчива к перепадам. Но если дефект все-таки реален, а кейс отказа возможен, композитную метрику можно разбить на изначальные метрики и посмотреть, была проблема или нет. Конечно, триггер воспроизведения проблемы не всегда получается сделать корректным с первого раза, поэтому проводится много циклов отладки.

ЗАКЛЮЧЕНИЕ

Риск отказов элементов системы есть всегда. И только мы оцениваем, какие из них приемлемые, а какие — нет. Руководствуясь этим, можно определить, соответствует ли система критериям отказоустойчивости и как отказы влияют на ее поведение. Осталось только решить, что делать с результатами. Ведь если они не будут использоваться, нет смысла тратить время на тестирование.

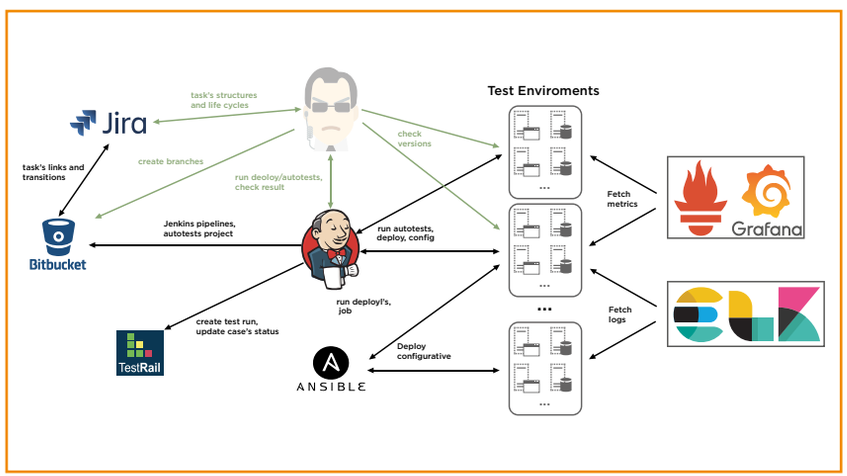
ТЕСТИРОВАНИЕ ОТКАЗОУСТОЙЧИВОСТИ
ВЫСОКОНАГРУЖЕННЫХ РАСПРЕДЕЛЕННЫХ СИСТЕМ
МИР Plat. Form

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ ПЛАТЕЖНОЙ СИСТЕМЫ

Мы изучили принципы построения тестирования отказоустойчивости, теперь давайте поговорим об интеграционном тестировании платежной системы.

13 систем, 11 интеграций между ними, десятки релизов в месяц — даже если каждый компонент работает хорошо, работают ли они как единая система?

Сначала рассмотрим проблемы, возникающие при организации интеграционного тестирования, а затем опишем решение, применяемое в Мир Plat.Form. В этом нам поможет руководитель команды интеграционного тестирования Владимир Мясников.



На схеме представлена итоговая экосистема команды интеграционного тестирования. Здесь можно увидеть:

- тестовые окружения, на которые настроен мониторинг и ELK-стек;
- Bitbucket, где располагаются проекты автотестов и автодеплоя интеграционной среды;
- Jenkins — практически сердце экосистемы, он запускает автотесты и деплой;
- TestRail, где хранятся тестовые кейсы и проставляется статус прогонов;
- Ansible, который является вспомогательным инструментом для деплоя;
- JIRA, где отображаются задачи по тестированию;
- СМИТ (Система Мониторинга Интеграционного Тестирования).

Процесс тестирования выглядит так, как показано на рисунке ниже:

Сначала остановимся на подводных камнях автотестирования и их преодолении.

ПРОБЛЕМЫ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ

КОНФЛИКТ БИБЛИОТЕК

Так как в проекте интеграционных автотестов переиспользуются проекты автотестов для тестирования отдельных продуктов, есть вероятность натолкнуться на конфликт библиотек (dependency hell). Например, в одном проекте используется библиотека одной версии, а в другом — та же самая библиотека, но другой версии, в которой какой-нибудь используемый метод уже отсутствует. Конечно, при сборке проекта получается stack trace «NoSuchMethodError».

Для решения такого рода конфликтов команда использовала инструменты Gradle.

Первый способ — исключить из зависимостей либо всю библиотеку, либо отдельный модуль:

```
compile (“ru.rep.at.borshTesting:2.1.0”) {

  exclude group: ‘com.fasterxml.jackson.core’, module: ‘jackson-databind’

  exclude group: ‘io.qameta.allure’

}
```

Второй способ — явно указать необходимую версию библиотеки при помощи resolutionStrategy:

```
configurations.all {

  resolutionStrategy {

    force ‘com.google.guava:guava:22.0’

    force ‘com.google.guava:guava-gwt:22.0’

    force ‘org.seleniumhq.selenium:selenium-api:3.6.0’

  }

}
```

Однако все равно могут возникнуть уникальные случаи, где такие подходы не действуют. В данных ситуациях необходимо договариваться со SDETами продуктов обновить у себя в проекте конфликтную библиотеку.

СТРУКТУРА ХРАНЕНИЯ FEATURES

Изначально feature-файлы в Мир Plat.Form были составлены в виде e2e-тестов с добавлением проверок сквозного функционала. Этот подход неоправданно повышает порог входа для новых сотрудников или переключившихся с другого проекта, а сами «простыни» ресурсоемкие в поддержке. Поэтому автотесты лучше разбить на интуитивно понятную структуру. На верхнем уровне разделить на e2e и интеграционные тесты. Потом e2e-тесты — на отдельные сервисы или стратегические проекты, а интеграционные тесты — на тестируемые системы, между которыми происходит интеграция.

В рамках отдельного сервиса или проекта мы выделяем бизнес-процессы, а в каждом из них — позитивные и негативные сценарии. В рамках интеграций — сквозной функционал, который тоже разбиваем на позитивные и негативные сценарии.

Для возможности гибкого запуска тестов каждый feature-файл мы отмечаем тегами, которые обозначают проверяемые системы в фиче, проверяемую сквозную функциональность и признак e2e-сценариев.

ОПИСАНИЕ ТЕСТОВЫХ ОКРУЖЕНИЙ

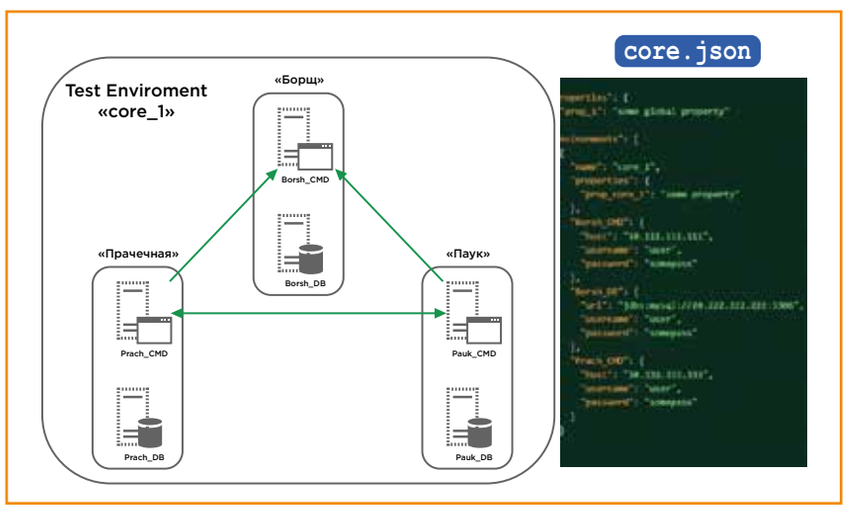
С ростом команды часто возникает необходимость масштабирования тестового окружения, представляющего из себя некий набор хостов с развернутыми системами (продуктами) и настроенными интеграциями между ними. Поначалу под каждого тестировщика выделялось персональное тесто-

вое окружение с развернутыми продуктами. Но это не позволяло им выполнять задачи параллельно. Например, нельзя было проверить только что разработанный автотест, потому что окружение было занято под прогон e2e-автотестов. Более того, не всем нужны все развернутые продукты в окружении. Поэтому встал вопрос унификации тестовых окружений для оптимизации их использования в различных условиях.

Тестовые окружения разбили на:

- core-окружения, где развернуты core-продукты;
- окружения под стратегические проекты;
- окружения для проверки бизнес-систем;
- e2e-окружения.

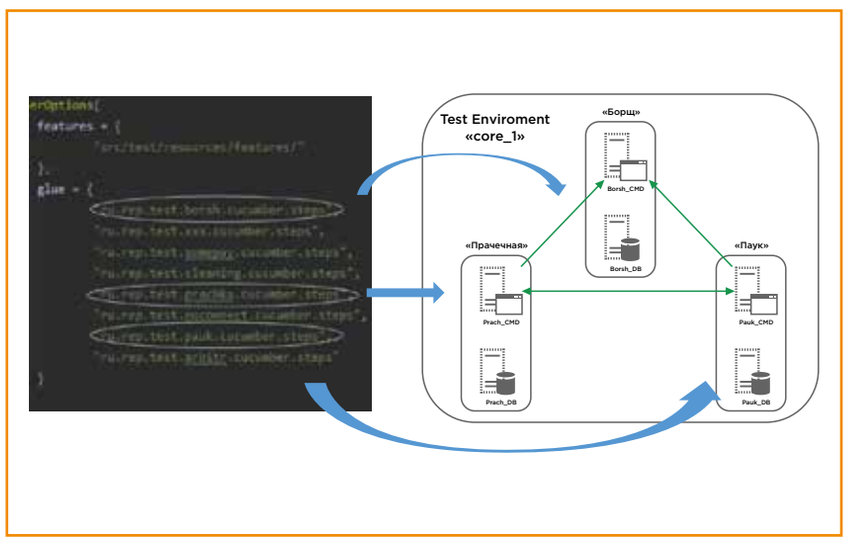
Помимо этого, на каждое окружение было сделано по несколько дубликатов для расширения возможностей параллельной работы. Тестовые окружения описываются с помощью JSON-структуры, «воспринимаемой» проектом интеграционных автотестов. В ней указаны ресурсы для работы с хостами, на которых развернуты продукты, и ресурсы для работы с базами данных продуктов.



GLUE ДЛЯ ОКРУЖЕНИЙ

Во время выполнения задачи по повышению производительности автотестов выяснилось, что их слабое место — инициализация переиспользуемых проектов перед запуском интеграционных автотестов. Например, при запуске автотестов на окружение, которое состоит из трех продуктов, должна происходить инициализация самого проекта интеграционных автотестов и трех проектов для тестирования соответствующих продуктов, но инициализировались все переиспользуемые проекты сразу.

Это было связано с тем, что в классе-раннере тестов в CucumberOptions в параметре glue перечислялись все возможные местоположения cucumber-шагов из всех переиспользуемых проектов.



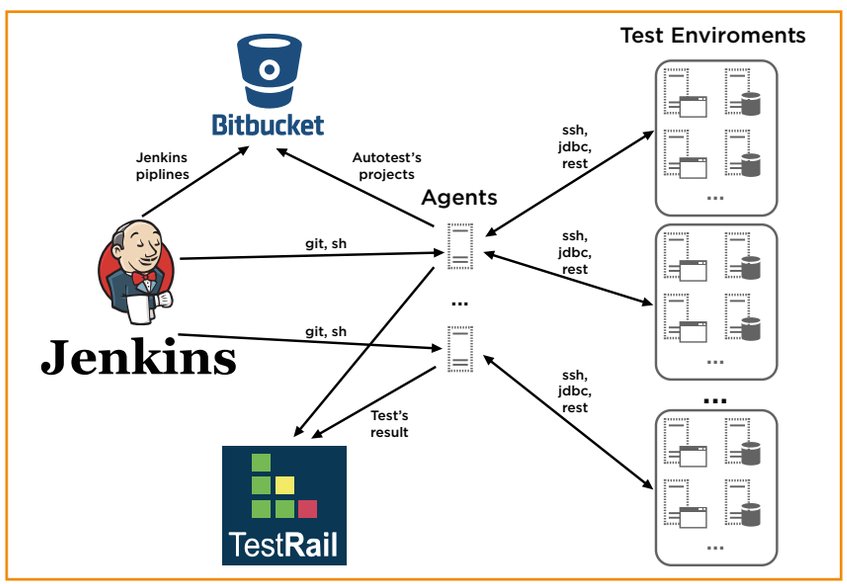
Конечно, мы можем передавать значение glue в параметре запуска. Но лучше добавить BeforeClass, который при помощи рефлексии подставляет в glue необходимые пути расположения шагов в зависимости от выбранного окружения. Это сокращает время тестирования примерно на 30%.

КОМФОРТНОЕ ТЕСТИРОВАНИЕ

Для удобства тестировщиков была оптимизирована технология их работы при помощи использования ELK-стека. Если раньше, при возникновении ошибки в e2e-тестировании, для локализации проблемной интеграции тестировщику приходилось «тонуть» в log-файлах всей цепочки систем — посмотреть лог первой системы, затем лог второй системы, а потом еще логи системы с микросервисной архитектурой. То теперь достаточно открыть Kibana, выбрать тестовое окружение и просмотреть агрегированные логи с сортировкой по времени.

Помимо этого, был настроен мониторинг стендов с использованием инструментов Prometheus, Grafana для своевременного реагирования, например, на заполнение дискового пространства. Это позволило повысить устойчивость работы модулей или сервисов на тестовой среде, т.к. на ней используются минимально необходимые для работы ресурсы.

Экосистема для проведения автотестов стала выглядеть так, как показано на рисунке ниже:



На этом по комфортной автоматизации все, но следует еще остановиться на внедрении автодеплоя.

АВТОМАТИЗАЦИЯ ДЕПЛОЯ ИНТЕГРАЦИОННОЙ СРЕДЫ

Изначально было три способа обновления систем:

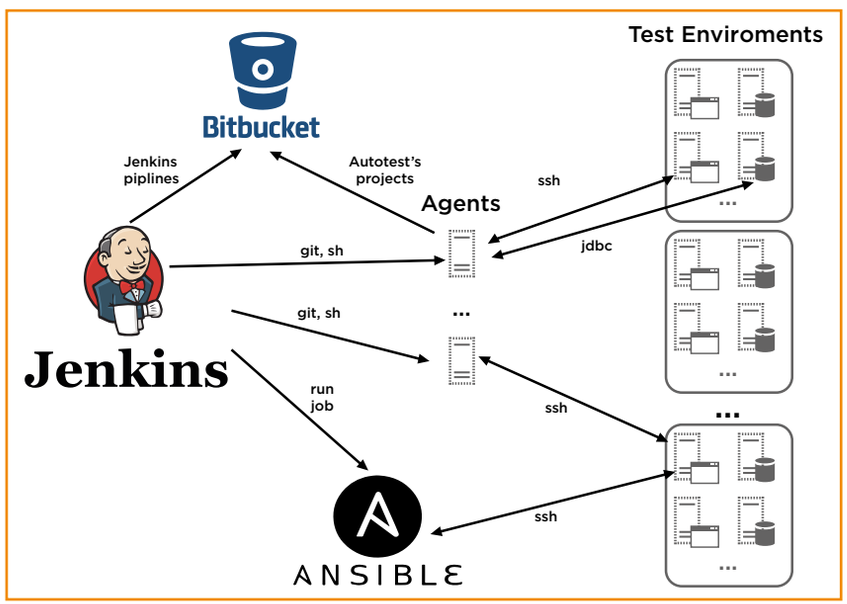
- скачать из Nexus артефакты и развернуть вручную на тестовом стенде;
- запустить Jenkins-пайплайн;
- запустить Ansible job.

Выбор способа зависел от обновляемой системы — для некоторых был подготовлен автодеплой, у некоторых — нет, но интеграционное взаимодействие всегда приходилось настраивать вручную. Чтобы получить единый флоу для деплоя систем и автоматической настройки интеграционного взаимодействия, нами была разработана Jenkins Shared Library, в которой применили фабричный паттерн проектирования.

Сначала определили основные этапы (stages) единого флоу автодеплоя. Каждый этап выделили в отдельный метод в groovy-интерфейсе: stopApp(), makeBackup(), deploy(). Затем для каждого продукта создали groovy-классы, импле-

ментирующие groovy-интерфейс со своей реализацией вышеперечисленных методов. И для выбора нужной реализации методов создали groovy-класс с фабрикой продуктов. После этого разработали пайплайн с разбивкой на этапы: «инициализация», «остановить приложение», «сделать бэкап», «задеплоить» и т.д.

При запуске автодеплоя происходит инициализация, в которой для выбранной системы «подхватывается» соответствующая реализация методов и устанавливается окружение, на котором необходимо обновить систему. Стоит отметить, что в данном пайплайне Мир Plat.Form есть этап «настройки интеграционного окружения», в рамках которого интеграционные параметры устанавливаемой системы задаются на основании JSON-файла с описанным окружением. В итоге экосистема для деплоя выглядит так, как показано на рисунке ниже:



Автоматизация деплоя позволяет тестировщику не тратить время на настройку отдельной системы и тестового окружения, а следовательно, интеграционное тестирование различных цепочек версий систем будет проводиться еще быстрее.

Остается только понять, какие цепочки нужно тестировать.

МИНИМИЗАЦИЯ РИСКОВ И РУТИНЫ

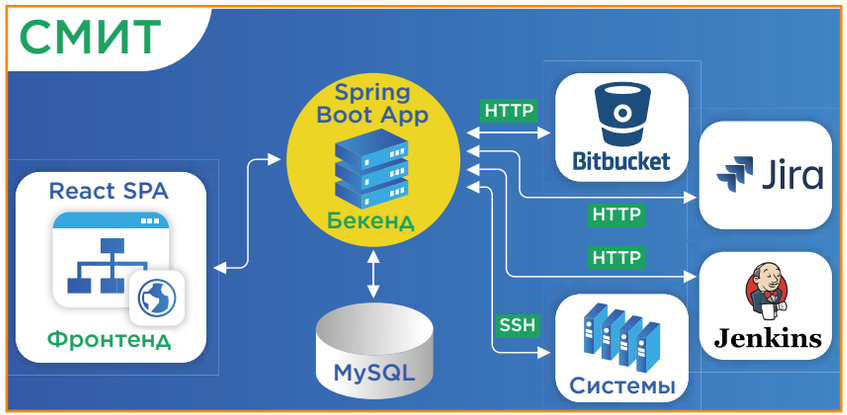
КАЛЕНДАРЬ РЕЛИЗОВ И СТРУКТУРА ЗАДАЧ

Для отслеживания актуальных версий продуктов и запланированных релизов появилась необходимость вести календарь в Confluence и табличку с перечнем продуктов и их текущих версий в продакшене. Чтобы команда понимала, какие интеграции необходимо проверить и с какой цепочкой версий продуктов, возникла потребность в создании Epic-структур релизов в Jira. Например, для условного релиза продукта «Прачка 1.45.0» создается Epic с задачами на анализ интеграционных изменений в рамках релиза, актуализацию автодеплоя, e2e-тестирование и тестирование отдельных интеграций между текущим продуктом и взаимодействующих с ним продуктами.

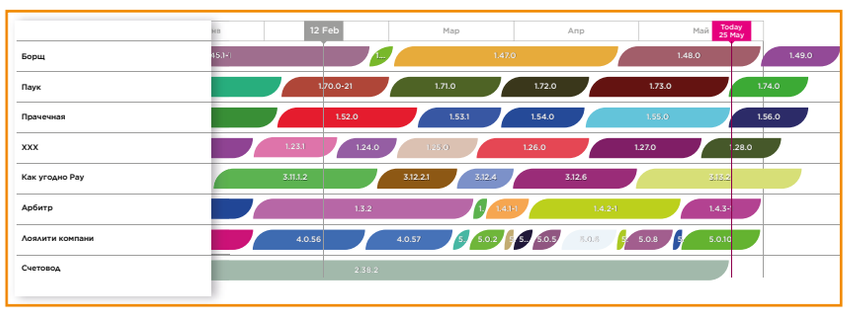
С одной стороны, все эти действия позволили улучшить понимание о тестируемых интеграциях, версиях систем и последовательности их выхода в продакшен. С другой — увеличилось количество рутины и осталась вероятность некорректного тестирования из-за человеческого фактора. Если дату релиза системы сдвигали, кому-то приходилось вручную править календарь и всю структуру в Jira, в том числе сроки выполнения задач. Перед тестированием интеграции стоило убедиться вручную, что окружение для тестирования состоит из нужных версий систем.

Для автоматизации подготовки к интеграционному тестированию релизов и минимизации рисков мы разработали систему СМИТ (Систему Мониторинга Интеграционного Тестирования). Она состоит из бэкенда и фронтенда, интегрируется непосредственно со стендами, где развернуты системы, с Jenkins, с Jira и Bitbucket. В качестве БД используется

MySQL, для бэкенда — Java Spring Boot, а у фронтенда — React. Все это завернуто в Docker и собирается при помощи Docker Compose.



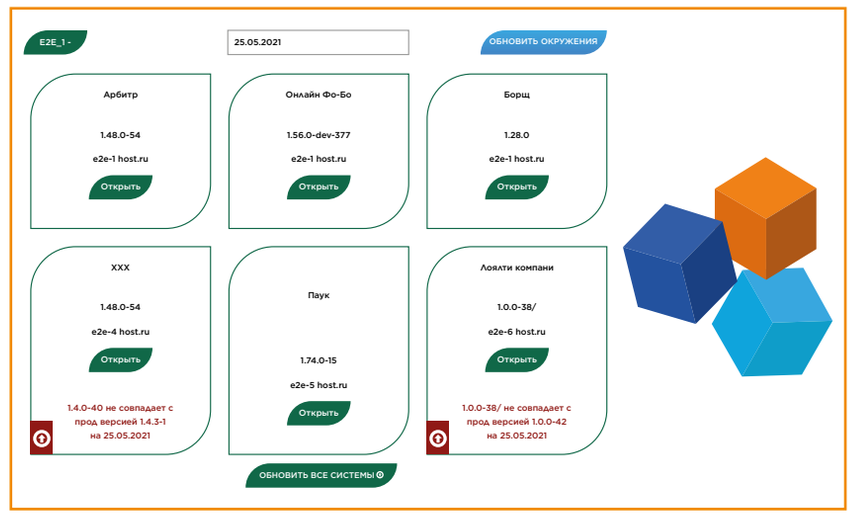
В этой системе ведется список тестируемых продуктов. У каждого есть отдельная карточка, где указывается его наименование, команда на получение версии, ресурс из JSON-файла с окружениями и ссылка на Jenkins job для запуска автодеплоя. Есть календарь релизов, в котором по кнопке можно зарегистрировать новую версию, указав, релиз какого продукта в какую дату установится в продакшен:



Стоит отметить, что во время регистрации новой версии релиза СМИТ создает Epic структуру в Jira и релизные ветки в Bitbucket. Также календарь позволяет просмотреть перечень версий продуктов на конкретную дату.

СОСТОЯНИЕ ТЕСТОВЫХ ОКРУЖЕНИЙ

Помимо вышеизложенного, в СМИТ есть страницы с визуализацией тестовых окружений, построенных на основании информации из JSON-файла с описанием окружений.



Оно состоит из 6 систем. Для каждой системы отображается, какая версия установлена на данный момент и на каких хостах развернуты продукты. Также на этой странице можно проверить соответствие версии системы на определенную дату и при необходимости запустить обновление до актуальной версии с настройкой интеграции.

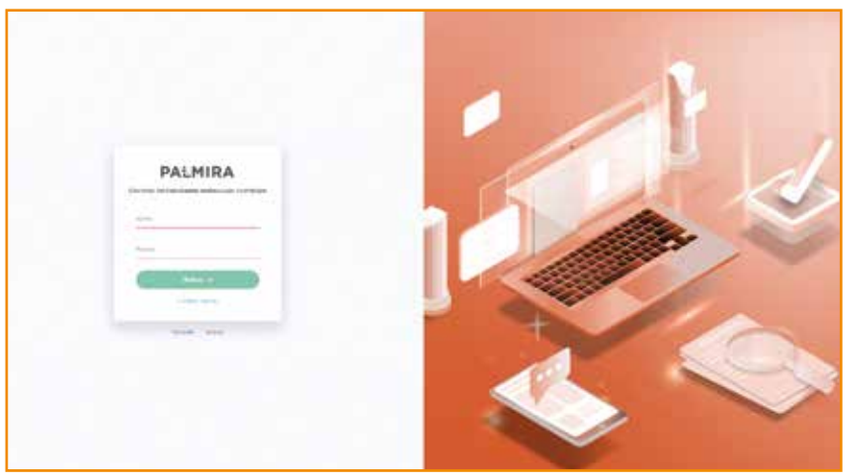
ЗАКЛЮЧЕНИЕ

К интеграционному тестированию на уровне систем необходимо подходить комплексно. Это позволяет развиваться «сферически» и лучше находить оптимальные решения. Например, команда интеграционного тестирования Мир Plat.Form теперь не только пишет автотесты, но и пробует себя в качестве разработчиков. Они продолжают искать пути оптимизации всех процессов на базе использования современных технологий и инструментов.

КАК АВТОМАТИЗИРОВАТЬ НЕВОЗМОЖНОЕ

Завершая тему тестирования, мы поговорим о решении, которое тестирует ключевую цепочку, — оплату. Как это сделать? Раздать каждому разработчику работающий и настроенный POS-терминал и набор кредитных карт? Не очень элегантное решение.

Разработчики Мир Plat.Form написали веб-сервис, который эмулирует процесс оплаты и позволяет легко и просто тестировать ключевую функциональность. Красивое решение, о котором и пойдет речь в этой главе. Подробности нам расскажет руководитель команды тестирования Валерий Богданов.

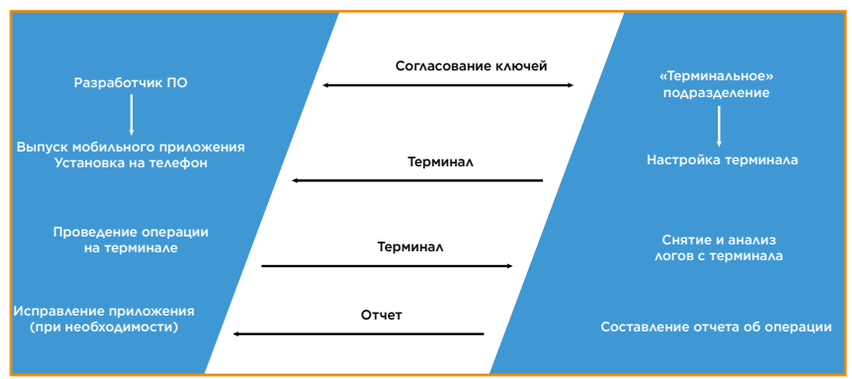


КЛАССИКА В ТЕСТИРОВАНИИ ТЕРМИНАЛОВ

Платежное ядро отвечает за логику взаимодействия телефона с POS-терминалом, то есть они должны общаться в строгом соответствии со спецификациями платежных систем. Помимо этого, оно участвует в принятии решения — отклонить или одобрить операцию. Еще оно отвечает за безопасность транзакций, так как именно в платежном ядре формируется платежная криптограмма.

Но протестировать его не так просто, как хотелось бы. В отличие от классических тестов, нужен платежный терминал (причем корректно настроенный), а он есть далеко не у всех. При этом не так много разработчиков и тестировщиков понимают, как телефон должен взаимодействовать с терминалом, и знают необходимые стандарты. Это, в общем-то, и не профиль разработчика.

Обычно для тестирования платежного ядра нужно сделать огромное количество операций. Классическая схема выглядит так, как показано на рисунке ниже:



Разработчику нужно выпустить какое-то приложение, согласовать ключи и операции на терминале с подразделением, которое за него отвечает. После этого дождаться, пока к нему приедет настроенный терминал, а он может ехать из другого города. Но даже после получения терминала и выполнения работ на нем, чтобы получить отчет по логам, его нужно снова отправить в департамент по терминалам. С учетом форс-мажорных обстоятельств — почта, курьеры, праздники — процесс может затянуться. Поэтому в первую очередь постарались оптимизировать этот процесс и сократить его до одного действия.

НОВАЯ СИСТЕМА ВМЕСТО КЛАССИЧЕСКОЙ

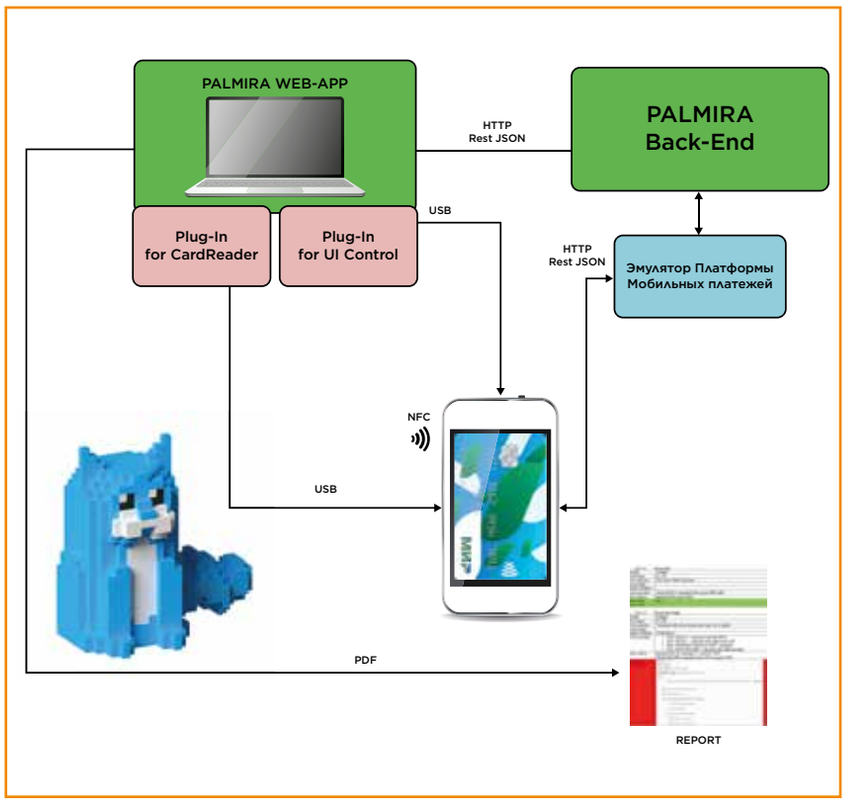
Еще на этапе аналитики был выявлен важный момент — интеграция кода после сборки может влиять на поведение

платежного ядра. Даже добавление систем защиты, не влияя на ядро напрямую, будет грубым вмешательством в код. Поэтому решили тестировать ядро после всех изменений и в максимально приближенной к релизной конфигурации.

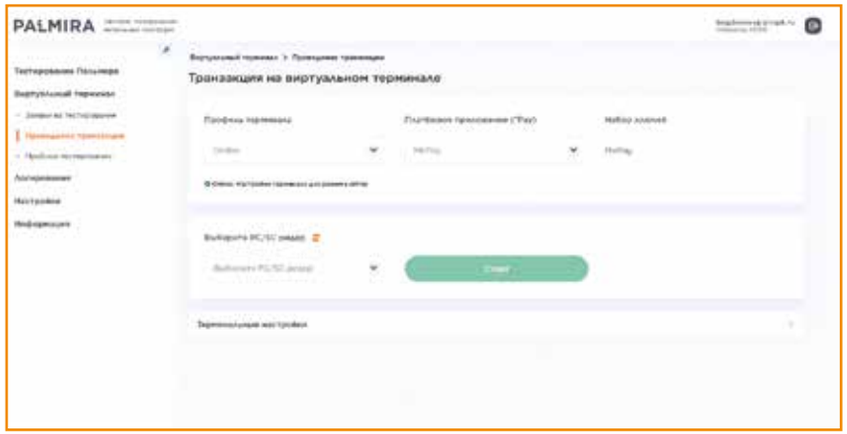
Помимо этого, на поведение платежного ядра могут влиять аппаратные особенности телефона и OS. Например, у одного из самых распространенных производителей телефонов между нашими командами пролетала команда от стороннего платежного приложения, предустановленного на телефон. У другого производителя была специфическая системная настройка, и без ее изменения вообще нельзя было добиться нормального информационного обмена с терминалом.

Можно было снова начать отправлять разработчикам наборы терминалов, настроенные под определенную операцию, но это бы вернуло тестирование к классической схеме потери времени на логистику. Поэтому начали разрабатывать сервис тестирования, которым бы мог пользоваться любой, даже без специфических знаний в области платежных систем и сложного оборудования. При этом в случае неуспешного теста в отчетах должно было быть достаточное описание некорректного поведения приложения.

С учетом всех требований был выбран формат веб-сервиса. Его преимущество в том, что практически на каждом ПК уже есть установленный браузер, который после небольшой доработки может выполнить роль клиентского приложения и связать PC/SC-считыватель пользователя с бэкендом системы. Это давало тестировщику возможность протестировать приложение на веб-странице с использованием простого PC/SC-считывателя. Чтобы подружить считыватель со страницей в браузере, использовали расширение. Большинство браузеров позволяют с помощью расширений производить информационный обмен через стандартные потоки ввода/вывода с нативными приложениями, например Native Messaging от Chrome. Для этого создали нативное приложение, которое занималось отправкой команд на PC/SC-считыватель с web-формы и возвращало ответ. Так, бэкенд системы мог непосредственно общаться по NFC с телефоном удаленного пользователя.

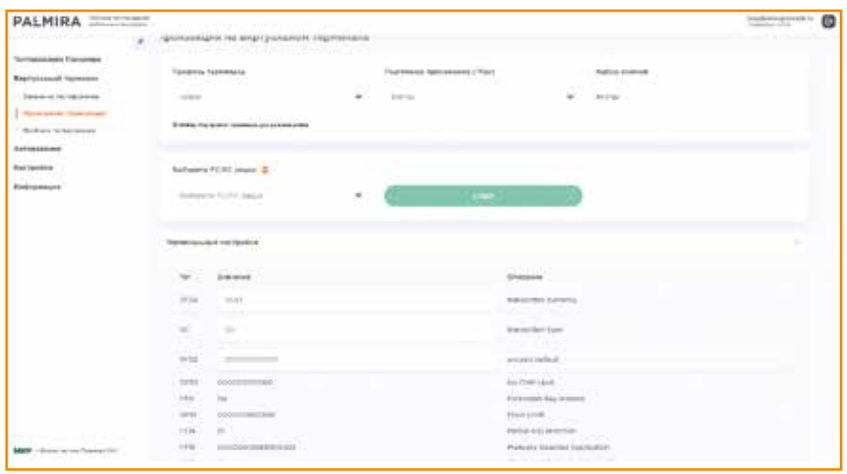


Помимо этого, был сделан настраиваемый виртуальный терминал, который в дефолтном состоянии полностью копировал функции POS-терминала из магазина. Его встроили в бэкенд системы и получили информационный тракт от «железного» PC/SC-считывателя к виртуальному терминалу. С его помощью пользователи могли бы удаленно совершать покупки. А тестировщик мог выбирать способ проведения оплаты и выставления суммы — на обычном POS-терминале или в банкомате.



После этого достаточно было приложить телефон к считывателю, чтобы имитировать операцию на настоящем терминале в магазине, и разработчик получал полный отчет по операции с распарсенными логами и подсвеченными ошибками, если такие были.

Изменение цикла разработки существенно сократило срок реализации ПО. Но оставалась еще одна проблема — платежное приложение партнеров может сильно отличаться от дефолтного.



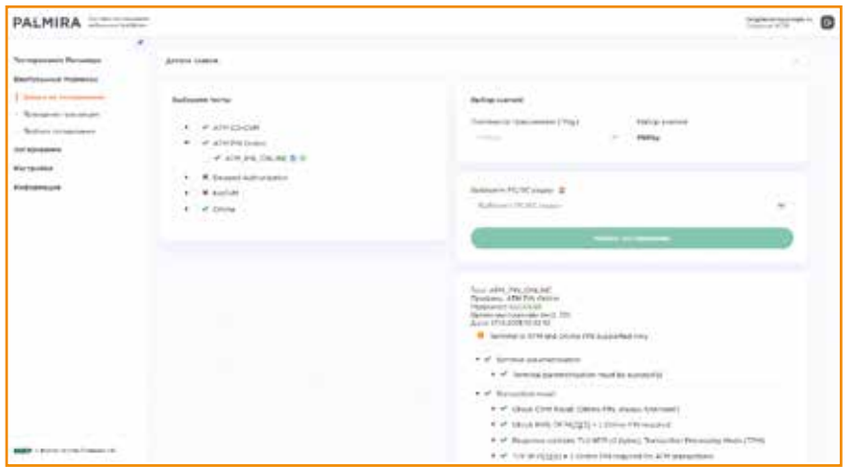
КАЖДОМУ СВОЕ

В партнерских платежных приложениях набор операций может быть разным. Например, изначально все варианты тестов были доступны разработчикам: для работы с банкоматами или для совершения покупок без подтверждения, но нужно было выключить функциональность, которая не планировалась или не использовалась. Поэтому для настройки системы реализовали ролевую модель, которая делила пользователей по организациям. Она включает в себя и роль администратора системы. Он определяет, какой набор операций сможет проводить та или иная организация, заводит ее профиль, добавляет операторов и администраторов уже внутри компании.

Администраторы могут заводить операторов (разработчиков и тестировщиков) только для своей организации. Помимо этого, добавили роли для службы безопасности, которые могут просматривать действия операторов и администраторов.

АВТОМАТИЗАЦИЯ

Mir Pay — собственная разработка Мир Plat.Form, поэтому для тестов его можно подключить к любым заглушкам, написав большой набор негативных кейсов. Но чем больше тестов, тем чаще придется прикладывать телефон к считывателю — что, конечно, неудобно. Чтобы это автоматизировать, было написано еще одно расширение. Приложение, с которым оно взаимодействовало, не отправляло команды на PC/SC-считыватель, а выполняло ADB-скрипты. Это позволяло управлять телефоном удаленного пользователя с бэкенда системы. Тестировщик заходил на страницу сервиса, выбирал нужные тесты и считыватель, нажимал на кнопку «Начать тестирование», клал подключенный к компьютеру телефон на считыватель, и больше его участие не требовалось.



При запуске первого теста в системе автоматически добавлялась платежная карта. На экране подключенного смартфона можно было видеть имитацию действий пользователя. После чего с заглушки бэкенда подгружался профиль карты и проводилась операция, при этом сверялись критерии выполнения теста. Затем все возвращалось в исходное состояние:

карта удалялась, данные и кеш чистились (если нужно) — и, независимо от того, прошел тест или нет, система переходила к следующему.

ЗАКЛЮЧЕНИЕ

С системой «Пальмира» полное тестирование платежного ядра с составлением отчета занимает около двух часов. Для этого были реализованы: виртуальный терминал, информационный обмен бэкенда с телефоном удаленного пользователя по NFC с помощью расширений браузера, управление удаленным телефоном скриптами с помощью расширений браузера и включение ролевой модели. Это позволило изменить цикл разработки мобильных платежных приложений и существенно сократить этап отладки платежного ядра.

СОВЕРШЕННОЙ СИСТЕМЫ НЕ СУЩЕСТВУЕТ, НО МЫ В ПРОЦЕССЕ

Мы поговорили с вами о верхнеуровневой архитектуре платежной системы, рассмотрели процессы тестирования. Какие еще аспекты работы с денежными средствами мы должны изучить особенно внимательно? Конечно, организацию безопасности.

Начнем с общих принципов построения безопасной системы, упомянем классическую модель безопасности, а затем изучим ее проекцию на работу команды Мир Plat.Form — от проектирования до эксплуатации.

Чтобы безопасность стала полезной, нужно разобраться, от кого и для чего мы защищаемся? Классическая безопасность определяется как сумма трех факторов: конфиденциальности, целостности и доступности. Что будет, если в платежной системе случится проблема с одним из них?

1. Конфиденциальность — никто не хочет, чтобы данные о его платежах и состоянии счета попали кому-то вовне.
2. Целостность — со счета не должен исчезнуть даже нуль.
3. Доступность — оплата платежной картой должна совершаться в любом терминале в любое время.

Разумеется, безопасность в платежных системах и нужна, и важна. Остается понять, насколько велик риск того, что она будет нарушена? Любая вероятность нарушения безопасности, которая больше нуля, на миллионах клиентов и карт даст огромный риск. Мы защищаемся не только от внешних, но и от внутренних злоумышленников. Не потому, что не доверяем нашим сотрудникам, а потому что нарушения безопасности часто допускаются случайно. Например, данные оставляют после отладки.

1. Существует много инструментов построения систем кибербезопасности. Представлены готовые циклы безопасной разработки Microsoft Security Development Lifecycle или цикл от Cisco. Разработаны PCI Security Standards и концепция SANS. Созданы CWE — база уязвимостей и OWASP — сообщество, которое расписывает, как надо делать безопасные системы. Существует целое движение DevSecOps.

Но готовый инструмент сложно применить без подготовки из-за того, что процессы обеспечения безопасности в организации всегда накладываются на текущие процессы. Первый вопрос, который встает, — как все это должно работать в каждой отдельной команде, если у вас 20 команд и 50 продуктов?

КЛАССИЧЕСКАЯ МОДЕЛЬ БЕЗОПАСНОСТИ

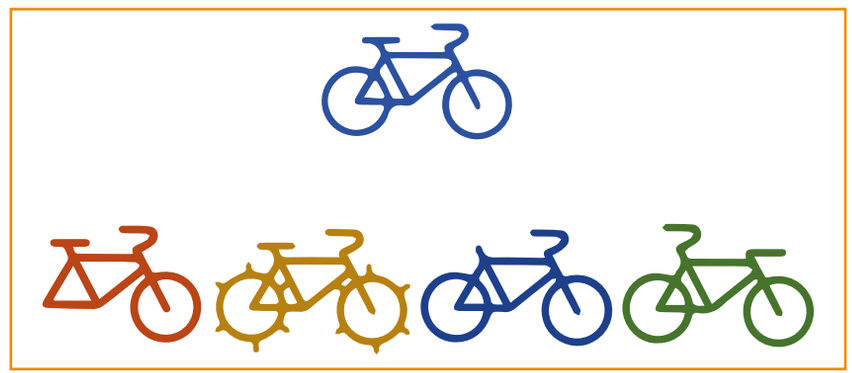
Представьте, что вы сделали очень безопасное приложение, но поставили его на уязвимый веб-сервер или подключили ненадежную СУБД — этим и может воспользоваться злоумышленник. Помимо этого, все обычно стоит на ОС или используется система виртуализации. Значит, их компрометация приведет к тем же последствиям. Еще могут быть проблемы с сетью, с процессами и людьми, и каждый из этих слоев: от кода и инфраструктуры до людей, которые пользуются приложением, должен быть защищен. А в хорошей системе нужны несколько средств защиты, чтобы даже после проникновения в один слой, по концепции Defense in depth, всегда были подстраховочные уровни так, как показано на рисунке ниже:

Есть стандартный цикл создания ПО:

- проектирование;
- разработка;
- сборка;
- тестирование;
- выпуск;
- эксплуатация.

Безопасность приложения должна обеспечиваться на всех стадиях его создания. Каждый артефакт, рожденный в процессе создания ПО, должен быть безопасным — начиная от архитектуры и заканчивая развернутым решением.

Представьте, что проектируете безопасный велосипед. Если на этапе архитектуры забыть про колеса — система не поедет. Если при реализации разработчик в коде использовал новую технологию и сделал что-то неправильно, получив «зубастый» код, — велосипед снова не поедет. И даже если мы весь проект реализуем хорошо, но используем opensource-компонент «седло», который окажется с недостатком, — ездить на велосипеде будет неудобно. Если что-то перепутать при сборке, велосипед начнет двигаться задом наперед.



Если на каждом уровне будут еще и проблемы с безопасностью, велосипед даже не выедет со склада, потому что запчасти растащат. То есть важны не только уровни, но и безопасность на них. Классическая схема, когда при тестировании сканируется код, а при выпуске — все окружение, и если нет недостатков, приложение считается безопасным, — дает только иллюзию безопасности. Такой процесс надо усовершенствовать. Например, вносить изменения во все этапы разработки систем.

Давайте пройдем по ним, чтобы посмотреть, как изменить классическую схему под себя.

ПРОЕКТИРОВАНИЕ

В первую очередь мы уходим от нормативных документов. Вместо этого делаем для разработчиков технические требования, в которых пишем, при обработке каких данных и что именно необходимо выполнять. Для этого нужен архитектор по ИБ. С ним разработчики могут обсудить, какие требования и стандарты покрывают конкретные риски, как и в какой ситуации их лучше применять. Это всегда более точечный подход, но если разработчикам все равно непонятны требования, то лучше обсуждать их всем вместе. Так возникающие сложности решаются гораздо эффективнее и быстрее.

Это позволяет изменить единый подход к безопасности — на рискориентированный. Например, можно создать пароль, который требует буквы, цифры, нижнее подчеркивание, верхнее подчеркивание и кучу спецсимволов, но при этом его длина только 6 символов. Можно дать возможность задавать только буквы и цифры, но при этом пароль должен быть не менее 12 символов. В обоих случаях объем символов для перебора и надежность одинаковые. Но второй подход более комфортный для пользователей. Он позволяет проявлять вариабельность при закрытии рисков, учитывать мнения всех сторон, а не просто выполнять требования безопасности.

РАЗРАБОТКА

Помимо этого, в рамках разработки можно отказаться от нормативных документов и перейти к описанию всех популярных тем в стеке своих технологий. Например, при работе с файлами указывать на то, что необходимо учитывать, чтобы работа была безопасной. То есть мы не пытаемся заставить разработчиков искать решение проблем безопасности, а показываем им, как это делать на примере кода, и говорим, что на нашем стеке есть такой метод фильтрации, а в Angular надо включить такие-то настройки.

Конечно, разработчик может предложить и свое решение, но должен согласовать его с безопасностью. Им известно больше нюансов и методик сравнения различных способов. На

этом этапе продолжаются обсуждения недостатков кода. Например, девиз отдела безопасности в Мир Plat.Form: «Договариваться — наш лучший подход внутри компании».

СБОРКА

Вместо того чтобы вручную просматривать известные базы уязвимостей и проверять каждую, лучше автоматизировать и проверять не только внешние зависимости, но и внутренние. Для этого тесты и проверки смещаются в этап сборки, чтобы баги вылезали еще до тестирования кода. Это и сканирование окружения тем же сканером кода, и проверки конкретных конфигураций на безопасность, и white-box-тестирование вместо black-box. Помимо этого, делается пентест (тестирование на проникновение), чтобы хотя бы на короткий промежуток смоделировать действия хакера. Это помогает понять, что мы не забыли поменять дефолтные пароли и не оставили открытые API.

Если заниматься всем этим на регрессе, в интеграции и во всех прочих видах тестов, то релиз может задержаться из-за исправлений, найденных багов. А так можно успеть исправить все то, что требует изменения кода на сборке, — и в тест пойдет более стабильная версия.

ТЕСТИРОВАНИЕ

В Мир Plat.Form используется много технологий, и для каждой есть свой сканер безопасности. Когда на скан выходит по 5 тысяч результатов, а 99%, если ставить из коробки, вообще ложные, работать с ними долго и неэффективно. Лучше написать собственный единый центр управления тестированием. Он умеет совмещать разные цепочки сканеров, чтобы создавать для каждого проекта свою связку. Несмотря на отличия команд в Git Flow, в ведении артефактов, веток или количестве релизов, все это можно унифицированно забирать для работы сканеров. Центр помогает отслеживать изменения в продуктах: ставить на мониторинг участки кода или файлы с критическим функционалом. Это позволяет уйти от тотального контроля, когда все проверяется заново, и на этапе тестирования

тоже перейти на рискориентированный подход, чтобы писать свой сканер под конкретный проект. Например, если мы не хотим, чтобы в проекте был файл с определенным содержимым. Это бывает очень важно при использовании нестандартных технологий.

Это помогает автоматизировать рутину. Представьте, что нужно протестировать API на доступность с 10 типами пользователей и 100 методами API. Можно сделать 1000 запросов и посмотреть, что же вернется, есть доступ или нет доступа и другие варианты. Можно решить в одну кнопку — выстроится матрица автоматических доступов, автоматически из кода выдернутся все методы API и все пользователи. И мы быстро и наглядно получим общую картину, кто к чему имеет доступ.

В Мир Plat.Form сейчас идут к тому, чтобы команды могли в IDE подключить линтер по безопасности и автоматически подсвечивать небезопасные конструкции, перейти от изолированных тестов конкретных систем к кросс-системным, интеграционным тестам по безопасности. Есть ситуации, когда одна система позволяет делать то, что не кажется опасным, и другая система это позволяет, а совместно они уже требуют внимания.

ЭКСПЛУАТАЦИЯ

На этом этапе должен обязательно быть мониторинг, отслеживание нештатных ситуаций и оперативное реагирование для максимально быстрого фикса.

АНАЛИЗ И УЛУЧШЕНИЕ

Даже изменение процессов полностью не исключает баги. Для этого необходимо обучать разработчиков. Они не обязаны быть экспертами по безопасности, но у них должен быть минимальный уровень. Мало завести баг в Jira и перекинуть на разработчика для исправления. Лучше описать в Jira его критичность и постараться объяснить все в личном общении. Если донести про последствия, в следующий раз этого уже не допустят. А еще такой разбор поднимает уровень разработчиков. Но если при анализе системы мы находим много недостатков, то можно провести тематический семинар для команды.

В Мир Plat.Form начали проводить обучающие игры в формате CTF (Capture The Flag). Играли на основе реальных недостатков, популярных недостатков из стека багов и тех, которые находили сами. Были и простые, и очень сложные задачи, но нашлись специалисты, которые решили все. Это подняло интерес к теме безопасности и разработки. Когда разработчики побывали в роли злоумышленников и через недостаток взломали систему, их понимание бага выросло. Наличие в командах специалистов, которые интересуются темой безопасности, позволило создать сообщество и непрестанно повышать культуру безопасного кода и архитектуры.

Для новых сотрудников ввели обязательный онбординг по обучению безопасности для достижения того самого минимального уровня знаний. Это помогает эффективнее выявлять самые простые баги. Онбординг тоже происходит в игровой форме с несколькими уровнями сложности. На каждом есть теория и задачи — все в единой тематике и с геймификацией. После анализа корреляции между пройденным обучением и выявленными багами оказалось, что CTF и обучающие игры работают лучше, чем тематические семинары. Стало меньше тех видов багов, которые рассматривались на семинарах.

Помимо этого, для улучшения самой команды продуктовой безопасности хорошо хотя бы раз в год привлекать внешних экспертов для тестирования. На основе их проверки можно скорректировать свои методики тестирования безопасности.

ЗАКЛЮЧЕНИЕ

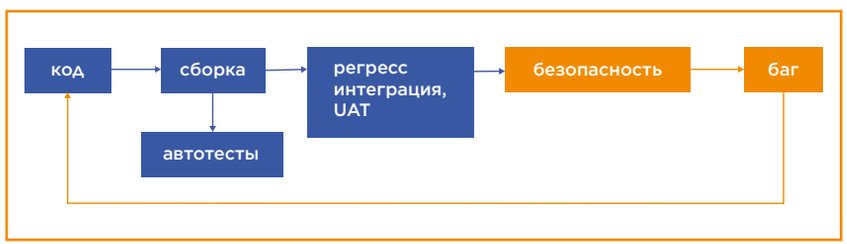
Чтобы безопасность стала более надежной, мало менять классический подход и встраиваться во все стандартные циклы создания ПО. Нужно наравне с непрерывным улучшением рекомендаций и практик внедрять новые технологии.

АВТОМАТИЗИРУЕМ ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ

Пойдем глубже — узнаем, как организовано тестирование безопасности, общие принципы которого мы рассмотрели в предыдущей главе. Да, мы поговорим о DevSecOps, о том, как оно устроено в Мир Plat.Form.

Цепочки сканеров и собственные сканеры, обработка результатов, отслеживание изменений, контроль уровня защищенности и автоматизация этого процесса. И даже автоматизация процесса обучения. Заместитель руководителя Центра программных решений Мир Plat.Form Алексей Бабенко поделится тем, как они это делают.

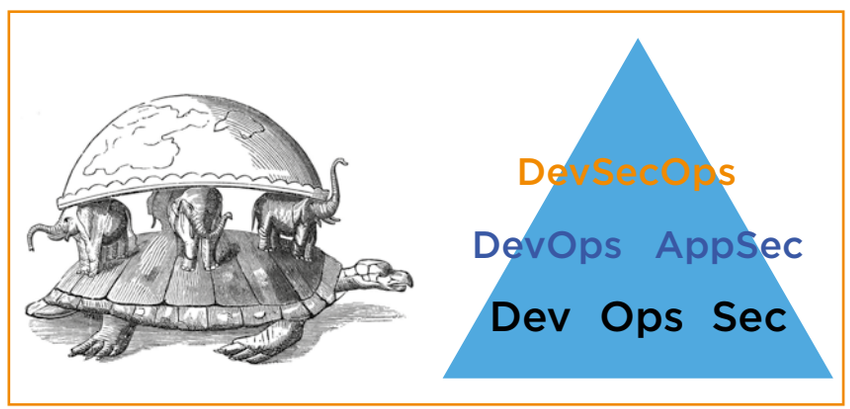
Начнем с того, как в Мир Plat.Form понимают DevSecOps. Чтобы безопасность не догоняла разработку: быстро меняющиеся релизы, сокращение Time to Market — и успевала за быстрыми изменениями, мало внедрить автоматизацию с сотнями сканеров. Нужно менять подход в целом. Поэтому в качестве основной функции DevSecOps определили «сдвиг влево». Это изменение места тестирования безопасности в стандартном цикле создания ПО, который выглядит так, как показано на рисунке ниже:



Обычно сначала создается код, выполняются сборка с использованием smoke-тестов, функциональное тестирование и только потом тестирование безопасности. Если последнее выявляет критичные баги, то вся цепочка запускается заново. Когда релиз проходит раз в год, такой подход еще

можно себе позволить, но если Time to Market сокращен до нескольких дней, так делать уже не получится. Поэтому и приходится делать «сдвиг влево», чтобы раньше начать поиск недостатков, как показано на рисунке ниже:

Когда классический подход меняется, команда безопасности уже не может дожидаться готового релиза, уходить на месяц и возвращаться с набором багов. Необходимо контролировать безопасность еще с написания кода. Например, подключая Linter'ы во время сборки и как только появляются первые артефакты. Для этого нужно автоматизировать проверку безопасности, чтобы быстрее реагировать на возникающие изменения. Но при таком подходе возникают свои подводные камни, как показано на рисунке ниже:



DEVSECOPS

DevSecOps строится на базовых понятиях: DevOps и продуктовая безопасность, которая основана на разработке, эксплуатации и безопасности в целом. Пока не выстроен нижний уровень пирамиды, сложно использовать более высокие практики. Очень трудно внедрять безопасность в автоматизацию, когда нет автоматизации или безопасности.

Если же классический DevOps начинает выполнять функции DevSecOps, то результаты со сканеров безопасности сразу отдаются на разработку и даже как-то обрабатываются. Но «из коробки» сканер дает тысячи срабатываний на большом проекте. В итоге разработчикам приходится разбираться, насколько баг критичный или же срабатывание является ложным, в то время как этим должны заниматься специалисты по безопасности.

Другая крайность — когда специалист по безопасности все автоматизирует и понимает, о чем идет речь, но у него не хватает рук, чтобы агрегировать и обрабатывать результаты. Он отсылает отчет разработчику, тот читает первые 20 страниц, видит одни ложные срабатывания, понимает, что дальше то же самое, и на этом DevSecOps заканчивается. Такая ситуация дает иллюзию ложной защищенности. Всем кажется, что безопасность обеспечивается, хотя фактически ее нет.

Третья крайность — обычная продуктовая безопасность без автоматизации и контроля изменяющихся процессов. Когда команда разработчиков существует отдельно от команды безопасности, которая находится в роли догоняющей. Она где-то что-то когда-то смотрит и, может быть, когда-то что-то находит, если до нее вообще доходят какие-то изменения.

ПУТЬ К БЕЗОПАСНЫМ ПРИЛОЖЕНИЯМ

Внедрение новых практик в Мир Plat.Form было обусловлено несколькими факторами. Прежде всего, ростом числа

систем и продолжающимся развитием старых. Тем, что новые релизы в продакшен выпускались каждые две недели, а иногда ежедневно. При этом тестировались новые технологии, которые тоже было необходимо отслеживать. Конечно, надо было расширять команду и быстро адаптировать новых разработчиков. Но гарантировать, что они будут соблюдать культуру безопасности разработки, как старые сотрудники, все равно было невозможно. Поэтому единственный вариант обеспечить безопасность в таких условиях — начинать автоматизировать все возможные процессы.

В Мир Plat.Form пошли по пути написания своего сервиса по управлению автоматизированным тестированием, чтобы облегчить работу со сканерами, сократить большое количество срабатываний и привести разрозненные результаты к единому виду.

ЦЕПОЧКИ СКАНЕРОВ

Сначала автоматизировали цепочки сканеров. Это позволило проверять каждую систему не всем пулом сканеров, а только выбранным набором. Это уменьшило количество срабатываний в целом и сократило время проверки.

Собственная оркестрация тестирования обеспечила быстрое подключение любых систем для проверки, несмотря на различия в проверяемых артефактах. Все результаты стали храниться в единой форме, их стало легко сравнивать или помечать как ложные.

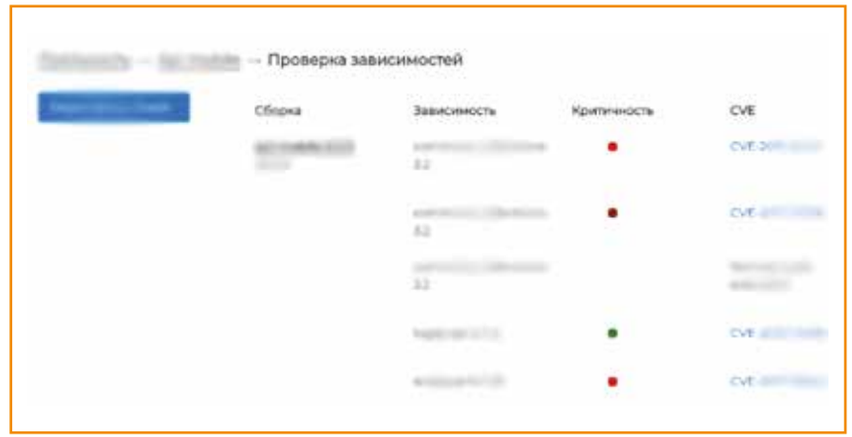
СОБСТВЕННЫЕ СКАНЕРЫ

В Мир Plat.Form стараемся отказаться от больших сканеров-комбайнов, которые умеют проверять все, но дают очень много ложных срабатываний, и стали делать свои сканеры для точечных решений. Например, сканер для тестирования API, чтобы проверять корректность реализации ролевой модели. Это можно сделать обычным Postman, но если у вас сотня методов и десяток пользователей, уже получается тысяча запросов. Проверять после каждого обновления долго, поэтому сделали автоматическую систему, как показано на рисунке:

Она не только проверяет доступность методов, но отлавливает ошибки и сохраняет текущее состояние. Достаточно один раз «прозвонить» все методы, сверить, насколько ролевая модель совпадает с документированным описанием, а потом смотреть различия. Если есть изменения, которые отличаются от предыдущей модели, их можно анализировать, если изменений не было, значит, модель корректна. Помимо этого, сканер показывает время выполнения метода, что важно при анализе атак типа «отказ в обслуживании».

Еще один пример точечного решения — сканер проверки приложений на Angular. Использовать большой сканер для анализа веб-приложений в авторежиме не всегда эффективно. Не каждый сканер может взять Single Page Application, и в целом анализ таких приложений в динамике достаточно трудоемкий и не всегда эффективный. Определенные моменты проще смотреть на уровне реализации проекта: кода и настроек. Это экономит время, повышает зону покрытия и количество найденных недостатков.

Другая важная возможность — перегрузка функций сканера. Например, есть сканер проверки зависимостей (см. рисунок):

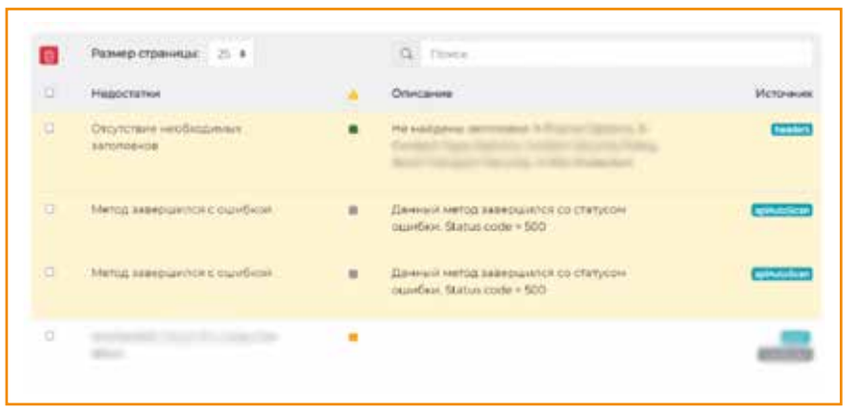


Он смотрит на внешние зависимости в коде, сравнивает с базой известных недостатков и проверяет, есть ли в ней версия этой зависимости. Зависимости бывают не только внешние, но и внутренние. Чтобы отлавливать и их, написали обертку для сканера, которая сначала строит перечень всех зависимостей проекта для текущей версии кода. Внешние зависимости сравнивает с внешней базой, а внутренние — с внутренней базой, которая ведется с версионностью и прочим всем необходимым. Это позволяет контролировать зависимости у ПО, которое уже вышло в продакшен. Например, если оно не обновлялось в течение месяца, а за это время в тех зависимостях, которые оно использует, выявили новые недостатки. ПО уже в продакшене, и если бы сканер зависимостей был встроен в пайплайн на уровне первичной сборки, то его бы не получилось проанализировать. А так сканер проверки зависимостей для каждой версии ПО строит таблицу и показывает, какие зависимости сейчас в продакшене. С его помощью можно не только протестировать актуальную версию в разработке, а еще просканировать версии в продакшене. Это позволяет узнать, что у ПО в продакшене появилась версия уязвимых зависимостей и нужно выпускать хотфикс.

РАБОТА С РЕЗУЛЬТАТАМИ

Сканеры «из коробки» дают примерно 95% ложных срабатываний. При этом разные сканеры находят одинаковые

результаты. Использовать их в автоматизации неэффективно. Разработчикам нужен результат в привычном виде, как в стандартном таск-трекере: с описанием недостатка, ожидаемым и текущим результатом и шагами по его устранению. Ни один сканер такой результат не даст, все равно потребуется ручная аналитика от специалиста. Поэтому результаты сканеров лучше собирать в единую таблицу, как показано на рисунке ниже:



Это позволяет помечать ложные срабатывания пакетными операциями, сканер это запоминает, и система их больше не показывает. Можно взять 10 однотипных срабатываний и на их основании завести метанедостаток. Получится стандартная запись в таск-трекере: критичность, описание, шаги по воспроизведению и прочее. Недостаток, найденный вручную, тоже можно завести, просто он не будет подкреплен автосредствами, но система все равно добавит его в таск-трекер.

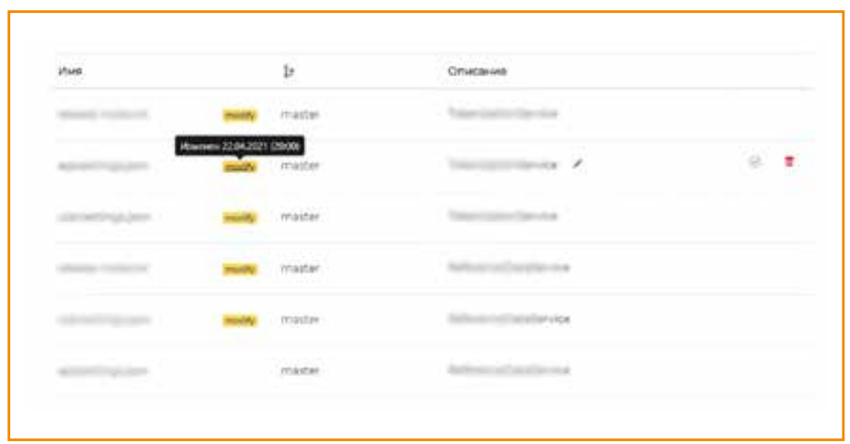
Когда разработчик берет недостаток из таск-трекера, разбирается, исправляет и переводит в статус «исправлен», система заново запускает сканеры и автоматически проверяет, исправился этот недостаток с точки зрения сканеров или нет. После этого специалист по тестированию безопасности может посмотреть код и подтвердить, что все исправлено. Такой подход значительно упрощает проверку и исправление недостатков.

РУЧНОЙ КОНТРОЛЬ

В тестировании безопасности полной автоматизации не бывает. С помощью сканера не получится найти логические баги, специфические недостатки связанные с криптографией или ограничением доступа. Без ручного контроля не обойтись, но некоторые этапы можно автоматизировать.

ОТСЛЕЖИВАНИЕ ИЗМЕНЕНИЙ

Части кода, касающиеся фильтрации и хранения данных, авторизации пользователей и логирования, не так часто изменяются в стабильной системе. Если, конечно, не вмешивается человеческий фактор. Руководители команд разработки знают, что перед выпуском релиза нужно пройти проверку безопасности. Но бывают ситуации, когда об этом забывают. Например, когда нужен срочный хотфикс. А еще разработчики могут поменять больше, чем рассказать. Или рассказать об одном, а поправить другое.



Чтобы решить эти проблемы, мы написали еще один сканер, который контролирует отдельные куски кода или отдельные файлы с критичной бизнес-логикой, влияющей на безопасность. Как только эти файлы изменяются, система оповещает об этом ответственных лиц вне зависимости от того, сообщили о них разработчики или нет.

При необходимости можно настроить систему так, что каждое изменение будет автоматически считаться недостатком. Это удобно для проверки уровня безопасности. Когда выпуск релиза согласуется разными сторонами, бывает сложно отследить, кто и какие изменения внес.

КОНТРОЛЬ БЕЗОПАСНОСТИ

Единая система учета недостатков позволяет проверять безопасность текущей версии на всех этапах. Например, у недостатков можно отмечать несколько состояний. Минорные недостатки не помешают согласованию и дальнейшему выпуску релиза. Для гарантированного исправления критичных недостатков вся история их состояний сохраняется. Это помогает понимать, что стоит на продакшене или в разработке, какие хотфиксы нужно или не нужно сделать и какие баги есть. С критичными багами команды разработки не выходят в релиз, а некритичные могут достаточно долго не исправлять, поэтому можно повышать критичность недостатка, чтобы он вошел в следующий релиз.

АВТОМАТИЗАЦИЯ ПРОЦЕССА

Помимо сбора информации и обработки результатов все равно остается много ручной работы. В Мир Plat.Form не стали останавливаться на автоматизации тестирования, а начали отслеживать скорость обработки результатов специалистами по тестированию безопасности и эффективность сканеров. Если время проверки начало расти, значит, специалисты не справляются и пора расширять штат или оптимизировать процессы. Если сканеры дублируют результаты, либо результаты нерелевантны или всегда дают ложные срабатывания, то они только съедают время и не приносят пользы и от них надо избавляться. Для этого ведется статистика эффективности. Все новые сканеры тестируются на релизе, который уже проверяли. Так можно увидеть, дают ли они новые результаты или только повторяют уже имеющиеся.

Разработчиков, тестировщиков и аналитиков не просто учат базовым правилам безопасности, этот процесс тоже автоматизируется. Сотрудникам неинтересно слушать о безопасности в целом, им нужно решение конкретных проблем. Поэтому всегда готовится точечное воздействие. Например, если у продукта возникают одни и те же недостатки, проводится тематическое обучение для команды, которая его разрабатывает. А потом оценивается эффективность обучения — повторяются ли те же самые недостатки в следующих релизах или нет.

ЗАКЛЮЧЕНИЕ

Применение DevSecOps-практик позволяет внедрять цепочки сканеров, которые снижают количество ложных срабатываний и быстрее решают локальные проблемы; автоматизировать обработку результатов; унифицировать отчеты и ускорять исправление ошибок. За счет отслеживания изменений снижается влияние человеческого фактора и попадание критических недостатков в релиз. Благодаря адаптивному обучению требованиям безопасности уменьшается рутинная работа и появляется больше времени на решение интересных задач.

ИНТЕГРАЦИЯ СЕРВИСОВ CI/CD В 2021 ГОДУ

Последняя глава книги собирает все, что мы уже знаем, в единую историю — это история про CI/CD (Continuous Integration/Continuous Delivery или Continuous Deployment).

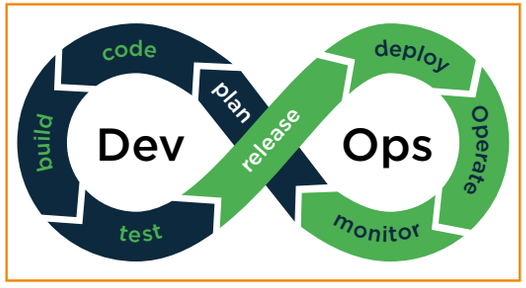
В Мир Plat.Form более 70 проектов, которые распределены по разным командам разработки. Как построить единую для всех систему? Как автоматизировать выделение ресурсов и сервисов?

Система построена по принципу Infrastructure as Code, о ее устройстве мы и поговорим в этой главе. В детали нас посвятит Игорь Николаев, руководитель инженеров автоматизации процессов разработки Мир Plat.Form.

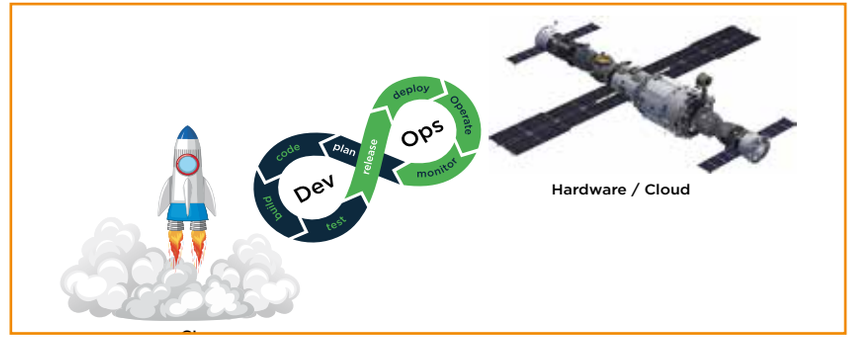
ЧТО БЫЛО

Иногда в Мир Plat.Form над одним из 70 проектов работают сразу несколько продуктовых команд, иногда, наоборот, одна команда разработки занимается несколькими проектами. Они могут относиться к разным департаментам, поэтому стандартизация подходов крайне важна.

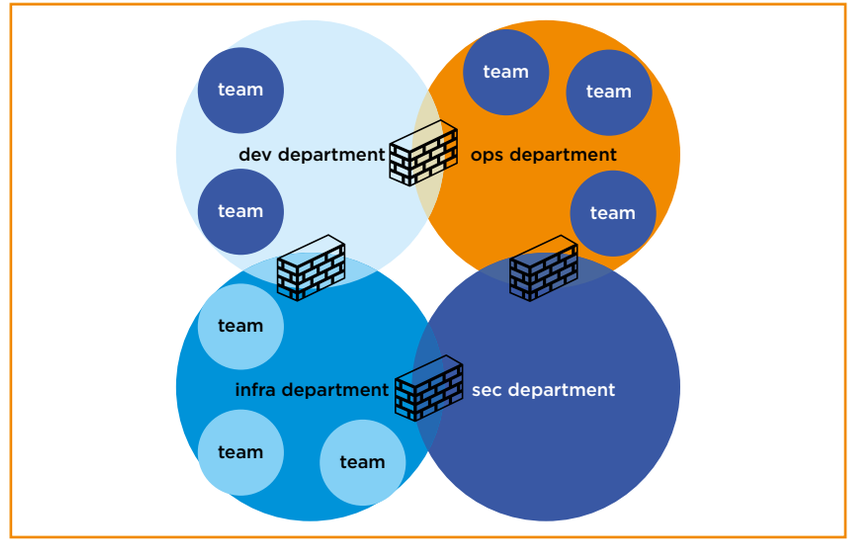
В целом проблема единообразия во всем, что касается разработки и практик DevOps, а также инструментов и сервисов CI/CD, стояла достаточно остро. Если бы можно было пользоваться публичными облачными решениями, данные задачи были покрыты инструментами и сервисами поставщиков облачных услуг. У них чаще всего есть готовые шаблоны для автоматизации инструментов и сервисов непрерывной интеграции и доставки.



В банковском секторе пользоваться облачными платформами нельзя в связи с повышенными требованиями к безопасности. Многие финансовые компании строят свои частные облачные решения, контейнеризацию и оркестрацию на собственном оборудовании. Тогда из классической схемы DevOps получится вот такая, как показано на рисунке ниже:



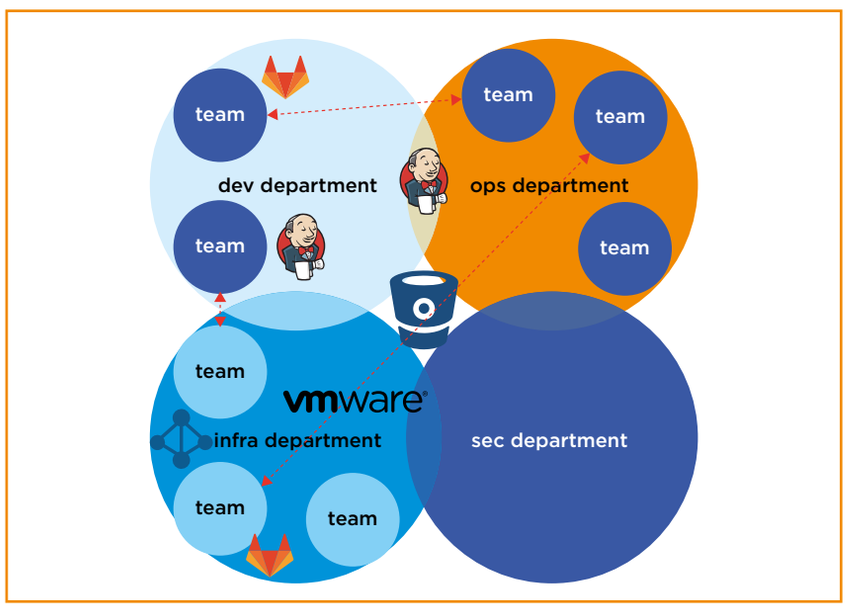
Структура Национальной системы платежных карт (НСПК) довольно сложная. Компания состоит из большого количества департаментов, однако в рамках процессов непрерывной интеграции основную роль играет часть из них.



1. Подразделение разработки состоит из команд разработки приложений и сервисов.
2. В подразделении инфраструктуры команды отвечают за поддержку домена, сети, физических серверов, системы виртуализации и Kubernetes.
3. В подразделении эксплуатации, соответственно, — за сопровождение промышленной эксплуатации.
4. Конечно, есть подразделение безопасности.

Им важно уметь эффективно взаимодействовать между собой, но из-за человеческого фактора и того, что задачи необходимо представить регламентами и контролировать, организовать это взаимодействие достаточно сложно.

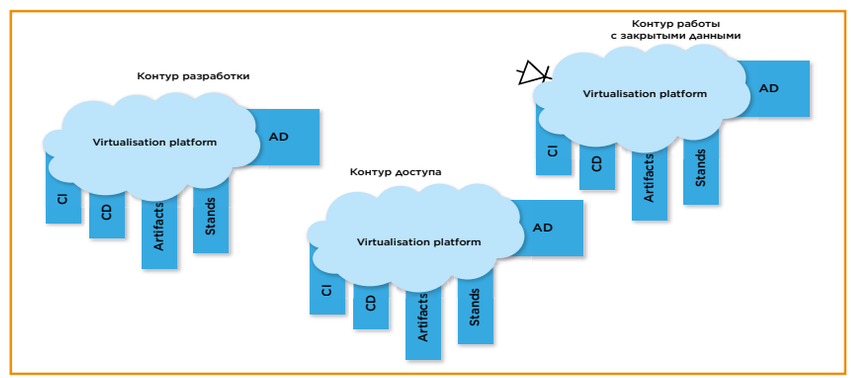
Из-за сжатых сроков реализации проектов было непросто сразу разработать стандарты для сервисов и подходов к организации CI/CD и DevOps. Команды могли использовать свои инстансы Jenkins, создавать специфичные группы AD или подходы, принятые в одном проекте, могли быть несовместимы с другим.



Сначала для упрощения задачи стандартизации выделили команду поддержки сервисов CI/CD, через которую планировалось наладить взаимодействие команд в разных подразделениях. Такой подход создал эффект «бутылочного горлышка» — всем подразделениям приходилось общаться через команду подразделения разработки, даже если задачи непосредственно к нему не относились.

ЧТО СТАЛО

Для решения данной задачи был разработан интерфейс взаимодействия между командами и департаментами по принципу Infrastructure as Code. Этот интерфейс объединил автоматизацию настроек, необходимых для работы с большей частью сервисов CI/CD и взаимодействие с сервисами других команд. Получился автоматизированный процесс для управления ресурсами платформы виртуализации, Active Directory для хранения пользователей и групп, и сервисов: Continuous Integration, Continuous Delivery, хранения артефактов (Nexus), Stands (стейджинг окружений и namespaces в k8s). Контуров достаточно много, но упрощенно их можно представить так, как показано на рисунке ниже:

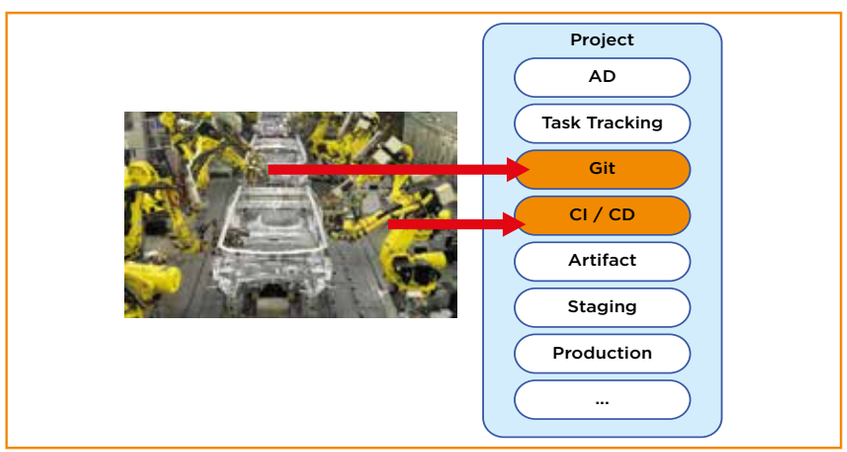


Эти контуры очень похожи, даже по виду получились «овечки Долли» — клоны со своими особенностями, которые «пасутся» в разных ЦОДах (центрах обработки данных).

ТИРАЖИРОВАНИЕ СЕРВИСОВ

Работая с одним проектом, достаточно управлять шаблонами сборки в рамках стандартного пайплайна, тестирования и публикации разрабатываемых сервисов. Но при большом количестве проектов мало тиражировать сами сервисы и их пайплайны, необходимо тиражировать целые проекты. Например, в Мир Plat.Form есть проект, который состоит из групп AD и пользователей с определенными правами. В качестве Task Tracking-системы используется Jira, в качестве хранилища исходного кода — BitBucket, инструменты CI/CD — Jenkins, хранилище артефактов — Nexus, окружение (стейджинг, продакшен).

При тиражировании сервисов чаще всего достаточно менять и добавлять только отдельные элементы, такие как новые репозитории Git, которые будут собираться с помощью CI/CD. Создавать новый репозиторий в Nexus не нужно, потому что он уже есть, как и новый стенд тестирования. Выглядит это так, как показано на рисунке ниже:



Представьте, что проект — это фабрика, на которой мы можем без лишних усилий создавать новые артефакты. Например, Git-репозитории. Тем более что при работе над одним проектом тиражировать что-то, помимо разрабатываемых сервисов, чаще всего не требуется. Но, если компания разрабатывает

большое количество различных проектов, которые не всегда напрямую связаны, требуется тиражировать именно проекты.

ТИРАЖИРОВАНИЕ ПРОЕКТОВ

Эта задача аналогична тиражированию сервисов, но включается в более глобальный пайплайн. Проекты становятся похожими друг на друга и могут включать в себя стандартные блоки, необходимые для выполнения текущих задач разработки.

Для тиражирования проектов необходимо автоматизировать работу со всеми сервисами, которые будут использоваться в пайплайнах. Нужно сделать стандартные шаблоны и дать возможность каждому проекту собирать из блоков требуемый пайплайн с возможностью добавления дополнительных сервисов или их замены в любой момент.

Если у вас много продуктовых команд, стандартизация блоков решает еще и часть сложностей при переходе сотрудника из одной команды в другую. Ему требуется меньше времени, чтобы адаптироваться к новому проекту и получить необходимые привилегии.

Системный подход к построению пайплайнов позволяет автоматизировать и другие процессы.

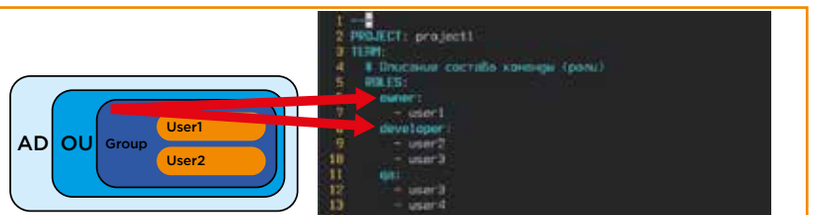
СИСТЕМНЫЙ ПОДХОД

Сущности в сервисах CI/CD Мир Plat.Form заводились в ручном режиме по заявкам, и получалось, что для проекта в AD-группу их заводил один специалист, в Bitbucket — другой, а в Nexus — третий. У проекта могли получиться разные именования и мог отсутствовать «общий знаменатель», так называемый ProjectKey. Для автоматизации пришлось бы делать таблицу совместимости, поэтому сначала проекты привели к общему именованию и везде добавили ProjectKey. Так, в любом сервисе всегда понятно, к какому проекту относятся те или иные сущности данного сервиса.

После этого занялись автоматизацией заявок на сущности в сервисах CI/CD. Заявка на предоставление ресурсов обычно требует заведения, согласования и исполнения. Получается, что на самые простые заявки, которые можно сделать очень быстро и автоматически, может уходить очень много времени. У нас появились карты проекта — YAML-файлы в Git-репозитории с плоской структурой, описывающей все ресурсы проекта.

Автоматизацию начали с Active Directory. Чтобы решить вопросы доступов, автоматизировали создание organization unit (OU) и групп доступа. Далее автоматизировали наполнение их пользователями, чтобы в дальнейшем этим группам выделять необходимые привилегии для автоматизированных сервисов.

Карта проекта для описания Active Directory выглядит как секция в YAML с названием «команды» и ролями в ней.




```
---
PROJECT: project1
TEAM:
  # Описание состава команды (роли)
  ROLES:
    owner:
      - user1
    developer:
      - user2
      - user3
    qa:
      - user3
      - user4

GLOBAL_PRIVILEGES: &global_privileges
  # Базовый набор привилегий для каждой роли в команде
  owner: [read, write, owner]
  developer: [read, write]
  qa: [read]
```


Команды могут называть роли по своему усмотрению. Внутри ролей находятся учетные записи пользователей, которые входят в состав команды. С помощью автоматизации легко формируется правильное название, в которое может быть включена дополнительная информация о принадлежности групп, например, удобно сортировать, добавив определенный префикс, чтобы было понятно, что группа обслуживается автоматизацией. В Active Directory это будет выглядеть как плоская структура директорий — organization unit (OU), наполненная группами безопасности (SG), в которые включены пользователи.

АВТОМАТИЗАЦИЯ

В качестве языка разработки системы автоматизации выбрали Python. Скорость работы автоматизации была не так важна, т.к. время выполнения задачи с разницей в 10 секунд или даже 1 минуту ни на что не повлияет, зато Python-код достаточно легко обслуживать. Также он входит в стек многих IT-специалистов.

В качестве оркестратора стали использовать Jenkins в связке с Jenkins Job Builder (JJB). Последний позволяет генерировать однотипные задачи из шаблонов и централизованно управлять ими в полностью автоматизированном режиме. Это удобно для автоматизации одинаковых задач, необходимых для обслуживания проектов.

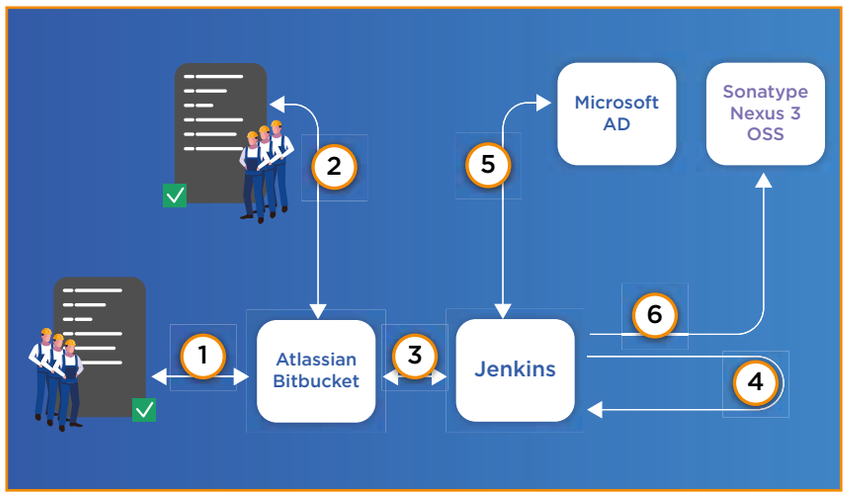
Для упрощения автоматизации каждую подключаемую систему описали как отдельный интегрируемый блок.

БЛОКИ

Каждый блок автоматизации обслуживает какую-либо систему: Active Directory, Bitbucket, Nexus и другие. Чтобы управлять блоками — включать, отключать или полностью заменять, — у них должен быть единый интерфейс. Для этого отлично подходит Git-репозиторий, который хранит в себе карты проектов.

Если упрощенно представить получившуюся автоматизацию, то процесс запроса получения сущностей или приви-

легий на проекте (или создание полностью нового проекта с получением всех сущностей) выглядит так, как показано на рисунке ниже:



Разработчики или заказчики, которым необходимо создать новый проект или внести изменения в существующий, описывают имеющийся YAML-файл проекта или копируют его из своего предыдущего проекта. Переименовывают сущности, добавляют людей в команды и делают pull request в Bitbucket (1).

Автоматизация проверяет pull request на валидность YAML и отправляет уведомление команде автоматизации, которая проверяет правильность заполнения. Если все верно, то pull request принимается (2), Bitbucket делает его слияние с мастер-веткой (3).

При слиянии изменений с мастер-веткой Git-репозитория Bitbucket уведомляет об этом Jenkins (4), и он запускает необходимые задачи (jobs), приводя их к описанному состоянию. Первыми обновляются технические задачи в случае их изменения, далее следует обновление общих задач CI/CD (5, 6).

Помимо запуска по перехвату в случае появления изменений в Git, сделали еще и ежедневное «принудительное» приведение системы к описанному состоянию. Это позволяет защититься от несанкционированного или неправильного ручного изменения конфигурации сервисов.

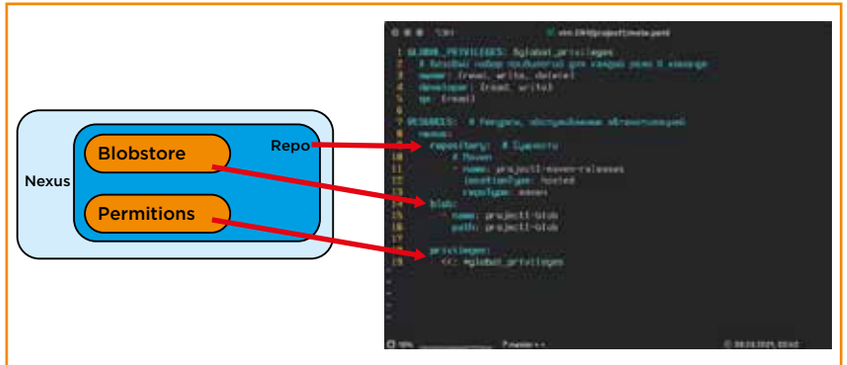
СЛОЖНОСТЬ

Система автоматизации устанавливает определенные требования к навыкам тех, кто ее поддерживает и разрабатывает, но дает значительную экономию времени. Автоматизируя процессы, мы можем проработать некоторые части так, что они будут использовать только настройки «по умолчанию», которые мы заранее предусмотрели, и скрывать некоторые сложные процессы от конечного заказчика. Прорабатывая интерфейс взаимодействия заказчика с системой, можно добиться упрощения процесса для конечного пользователя и сделать задачу шаблонной. Например, построить цикл разработки так: написать код, собрать приложение, протестировать его, проверсионировать и опубликовать в хранилище артефактов для дальнейшего переиспользования, например, готовые docker tools, такие как jdk с gradle. Помимо этого, можно почистить сборочное окружение и сделать мониторинг и логирование.

При этом разработчик освободится от задач версионирования, постановки, мониторинга и логирования. Эти процессы должны иметь понятный для него интерфейс подключения и быть максимально типовыми. При получении достаточного количества блоков автоматизации появляется возможность делать типовые процессы сборки артефактов. Для получения типовой задачи в Jenkins все необходимое можно взять из карты проекта, остается уточнить, какой версией Java и Gradle пользоваться для сборки.

За простым интерфейсом взаимодействия с автоматизацией скрыты детали, которые не всегда учитываются при старте нового проекта и могут привести к дополнительным сложностям в дальнейшем. Чтобы избежать этого, разработчики часто готовы пойти на компромисс, чтобы не

тратить время на решение типовых задач, решаемых автоматизацией. При этом надо оставлять возможность переопределения переменных для автоматизации и замену дефолтных значений в случае появления объективно обоснованных нестандартных задач. Например, YAML-карты проекта для хранилища репозиториев Nexus в Мир Plat. Form реализованы так, как показано на рисунке ниже:



Для базовой автоматизации Nexus достаточно передать системе автоматизации название, формат репозитория и location type. Для работы с API необходимо значительно больше параметров, но значения по умолчанию мы продумали заранее, за счет этого упрощается и уменьшается размер самой карты проекта.

В примере видно, сколько параметров необходимо для создания репозитория и насколько их можно сократить при использовании стандартных значений.

Помимо этого, YAML позволяет делать ссылки и при необходимости обращаться к ним, чтобы не переписывать секции описания.

GLOBAL_PRIVILEGES — абстракция над любыми привилегиями, которые могут быть в сервисе. Для нормализации значений дефолтных сущностей (в примере это owner, developer и QA) автоматизация каждого сервиса подразумевает проработку стандартных наборов привилегий, которые заменяют нормализованные упрощенные значения.

ЗАКЛЮЧЕНИЕ

Важно выбрать интерфейс для работы с сервисами и пользователями, скрыв сложные технические решения за автоматизацией, чтобы упростить весь процесс взаимодействия с автоматизацией. Проекты полезно делать шаблонными и стандартизированными. Это позволяет не только упростить многие процессы, но и легко контролировать все сервисы, покрытые автоматизацией.

ЗАКЛЮЧЕНИЕ

Если вы прочитали эту книгу, скорее всего, вы строите свою карьеру в IT. Мы вместе с вами работаем в одной из самых востребованных отраслей мира и не понаслышке знаем, что она непрерывно меняется и развивается.

Чтобы сохранять лидирующие позиции, обеспечивать максимальную надежность и доступность Национальной системы платежных карт, Мир Plat.Form каждый день оптимизирует процессы, улучшает практики, внедряет инновационные разработки и гарантирует безопасность процессов. Мы меняем классический подход и постоянно модернизируем наши сервисы, используя современные технологии, не накапливая технический долг.

Мир Plat.Form работает над масштабными проектами, которые действительно полезны для людей. Решения, представленные в этой книге, актуальны для 2021 года. Если вы читаете ее позже, знайте: мы уже на шаг впереди.

Хотите развивать финансовую индустрию в России, делать жизнь в своей стране проще и удобнее?

Сканируйте QR-код и присоединяйтесь к команде Мир Plat.Form!