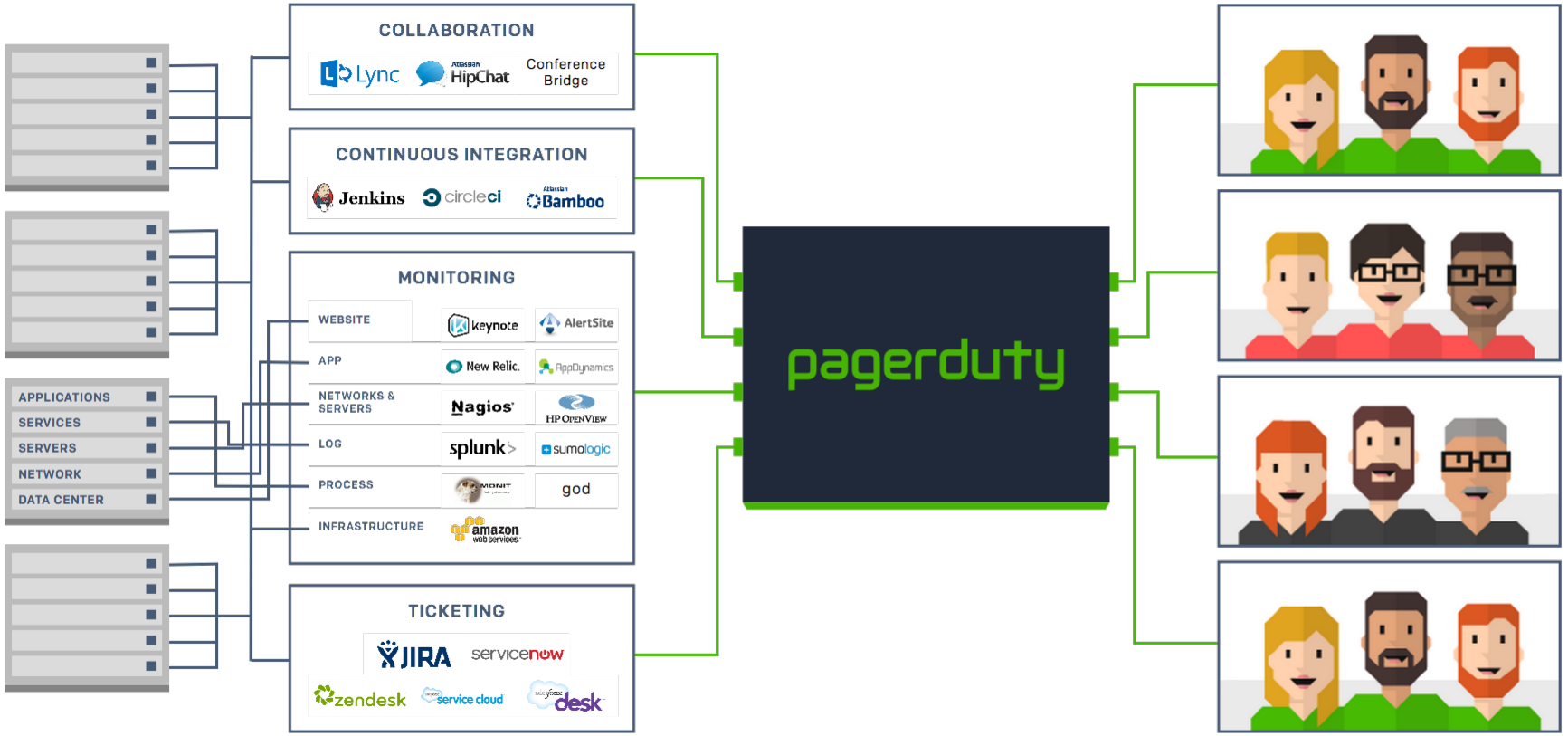
pagerduty

WatchDog

Using Scala for end-to-end functional testing

@klprose @pagerduty

What is PagerDuty?

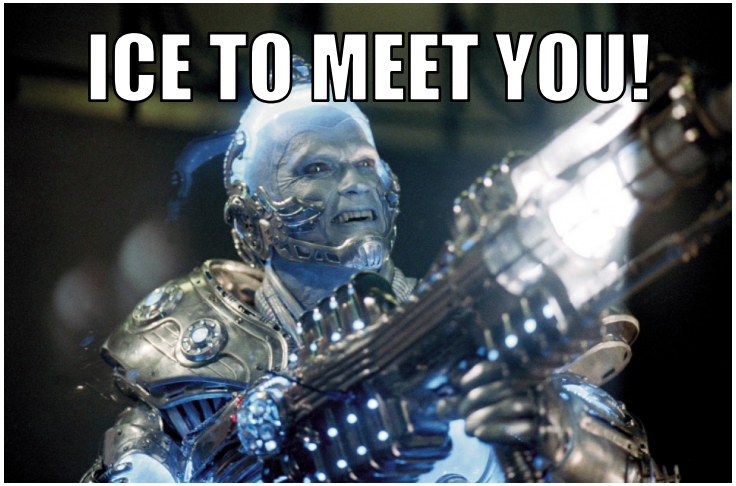


Reliability at PagerDuty

- We have to be up when your infrastructure is not
- Our customers trust us to reliably deliver alerts
- We ❤️ reliability
- We use a multi-DC SOA so that even a DC outage does not stop alert delivery

Reliability at PagerDuty

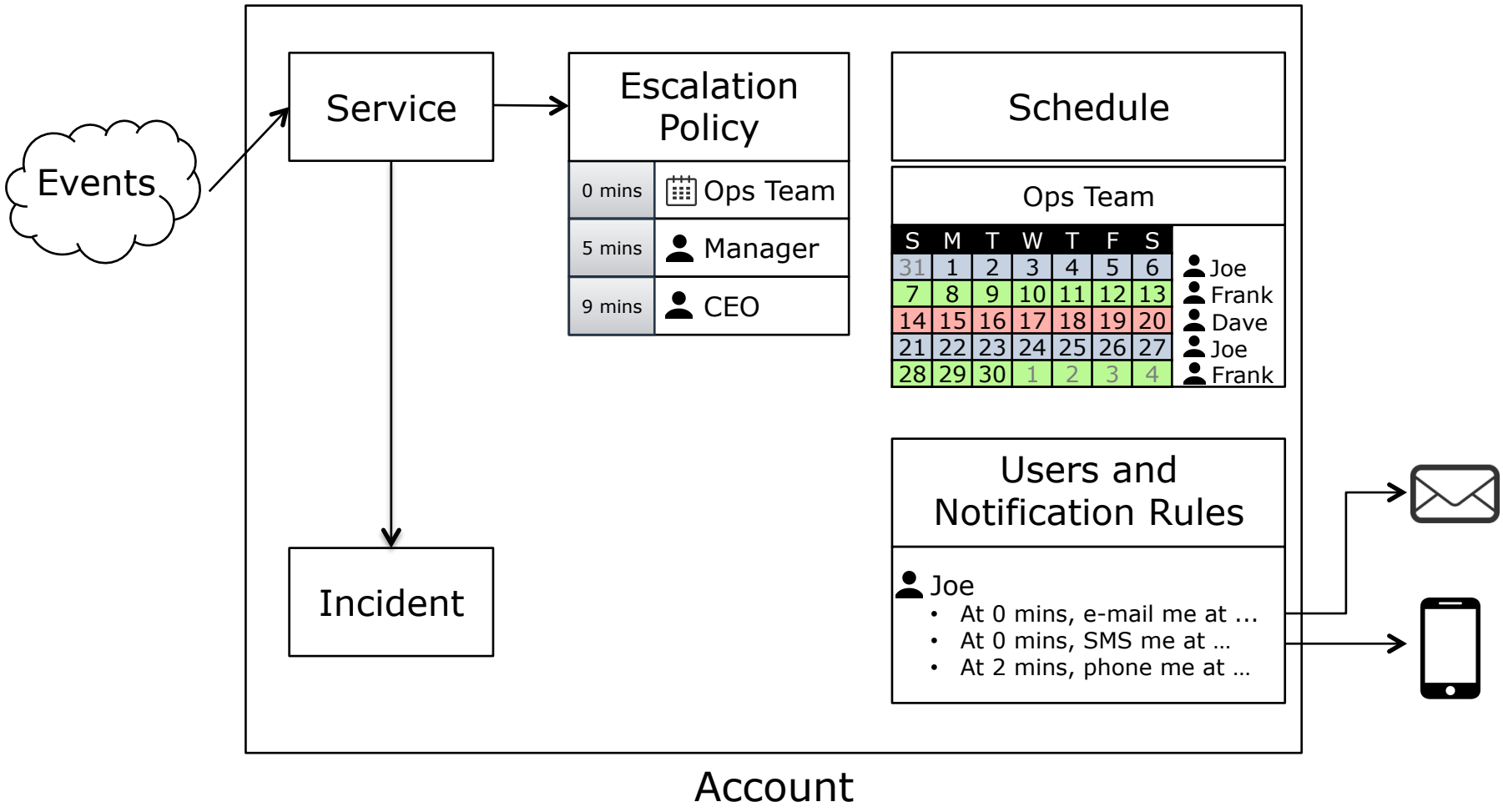
- Q3 2014: Two silent SEV-1s
- Exposed gaps in our testing and deployment procedures
- Code Freeze until fixed



Improving Reliability

- Solution: Write an end to end functional test suite
- Behave like a customer
  - a really diligent customer that uses lots of PD functionality
  - and uses it really, really often
- Shout from the rooftops if PD doesn't work as expected
- Basically: Run tests against PD and alert on failures
- Important:
  - Test the entire system working together
  - Production is the gold standard

Aside: PagerDuty's Data Model



First Attempt at a Test

```
"Sending an event" should "create an incident" {
  // 1. Provision an account... somehow
  val account = ???

  // 2. Make a bunch of resources on the account
  val user = account.mkUser
  val schedule = account.mkSchedule
  schedule.add(user)
  val escalationPolicy = account.mkEscalationPolicy
  escalationPolicy.addEscalationRule(EscalationRule(schedule))
  val service = account.mkService(ServiceType.Generic, escalationPolicy)

  // 3. Create a generic event
  val eventDescription = "Server on Fire"
  val triggerEvent = GenericEvent(
    serviceKey = service.key,
    description = eventDescription,
    eventType = "trigger");

  // 4. Send the event to PagerDuty
  triggerEvent.send

  // 5. Wait for an incident
  val incident = waitForIncidentToBeCreated // somehow?

  // 6. Validate properties of the incident
  incident.state shouldBe "trigger"
  incident.description shouldBe eventDescription
}
```

Yes how should accounts be provisioned?

This is very boilerplatey

Potentially huge API

Yes, how should a test specify which incident it's waiting for?
Is waiting synchronous?

pagerduty

Observations and Considerations

- We’re using ScalaTest
- Synchronous polling is OK
- This is against prod; be mindful of load

Dealing with Issues

1. Account provisioning
2. Setting up account entities (e.g., services, EPs)
3. Waiting for occurrences (e.g., a new triggered incident)

Account Provisioning

- Started with

```
"Sending an event" should "create an incident" {
  // 1. Provision an account... somehow
  val account: Account = ???
```

- Responsibility of creating an account is on test author.
- Use loan-fixture method instead [1]

```
trait Account { def subdomain: String; def authToken: String }
def withAccount[R](test: Account => R) = ???

"Sending an event" should "create an incident" in withAccount {
   account: Account =>
     // do stuff with account...
 }
```

[1] - http://scalatest.org/user_guide/sharing_fixtures#loanFixtureMethods

Account Provisioning

ScalaTest docs:

> *"A loan-fixture method takes a function whose body forms part or all of a test's code. It creates a fixture, passes it to the test code by invoking the function, then cleans up the fixture after the function returns."*

Let's see it in action

```
trait AccountManager {
  def acquireAccounts(num: Int): Set[Account]
  def releaseAccounts(accounts: Set[Account])
}

// The loan fixture: mix this in to your tests
trait AccountAcquiring {
  def withAccount[R](test: Account => R)
    (implicit accountManager: AccountManager): R =
    runTestsAndReleaseAccounts(1, accountManager,
      { accounts: Set[Account] => test(accounts.head) })

  def withAccounts[R](num: Int)(test: Set[Account] => R)
    (implicit accountManager: AccountManager): R =
    runTestsAndReleaseAccounts(num, accountManager,
      { accounts: Set[Account] => test(accounts) })

  private def runTestsAndReleaseAccounts[TestResult]
    (numAccounts: Int,
     accountManager: AccountManager,
     test: Set[Account] => TestResult): TestResult = {

    val accounts = accountManager.acquireAccounts(numAccounts)

    try {
      test(accounts)
    }
    finally  {
      accountManager.releaseAccounts(accounts)
    }
  }
}
```

Account Provisioning: Load Considerations

We can either:

1. Create new accounts each time
2. Re-use accounts from a pool

100s of tests each run every 5 minutes

=> 28K+ accounts created per day

Dealing with Issues

1. Account provisioning
2. Setting up account entities (e.g., services, EPs)
3. Waiting for occurrences (e.g., a new triggered incident)

Setting up Account Entities

- We started with:

```
// 2. Make a bunch of resources on the account
val user = account.mkUser
val schedule = account.mkSchedule
schedule.add(user)
val escalationPolicy = account.mkEscalationPolicy
escalationPolicy.addEscalationRule(EscalationRule(schedule))
val service = account.mkService(ServiceType.Generic, escalationPolicy)
```

- Lots of boilerplate.
- As a test author, would be nice to specify only what I need.

Setting up Account Entities

- Use the builder pattern to encapsulate setting defaults

```
"Sending an event" should "create an incident" in withAccount {
  account: Account =>

  val b: ServiceBuilder = new ServiceBuilder
  val service: Service = b.createOn account
}
```

Setting up Account Entities

- Builders are chainable

```
"Sending an event" should "create an incident" in withAccount {
  account: Account =>

  val b: ServiceBuilder = (new ServiceBuilder).
    autoResolveTimeout(45.minutes)
    acknowledgementTimeout(15.minutes)
  val service: Service = b.createOn account
}
```

Setting up Account Entities

- Refactor a singleton for nicer tests
- Could have mixed in a trait

```
// Builder.scala
object Builder { def builder = new BuilderPicker }
sealed class BuilderPicker {
  def service(serviceType: Service.Type) = new ServiceBuilder(serviceType)
}

// In test
import Builder.builder
"Sending an event" should "create an incident" in withAccount {
  account: Account =>

  val service: Service = builder.service(Service.Generic).
    autoResolveTimeout(45.minutes)
    acknowledgementTimeout(15.minutes)
    createOn(account)
}
```

Setting up Account Entities

- Setting other PagerDuty entities (e.g., an escalation policy)

```
// Builder.scala
object Builder { def builder = new BuilderPicker }
sealed class BuilderPicker {
  def service(serviceType: Service.Type) = new ServiceBuilder(serviceType)
  def escalationPolicy = new EscalationPolicyBuilder
}

// In test
"Sending an event" should "create an incident" in withAccount {
  account: Account =>

  val ep: EscalationPolicy = builder.escalationPolicy.createOn(account)

  val service: Service = builder.service(Service.Generic).
    autoResolveTimeout(45.minutes)
    escalationPolicy(ep)
    createOn(account)
}
```

Setting up Account Entities

- Recursively and lazily create all entities with `createOn`

```
// Builder.scala
object Builder { def builder = new BuilderPicker }
sealed class BuilderPicker {
  def service(serviceType: Service.Type) = new ServiceBuilder(serviceType)
  def escalationPolicy = new EscalationPolicyBuilder
}

// In test
"Sending an event" should "create an incident" in withAccount {
  account: Account =>

  val epBuilder: EscalationPolicyBuilder = builder.escalationPolicy

  val service: Service = builder.
    service(Service.Generic).
    escalationPolicy(epBuilder). // Override for EscalationPolicyBuilder
    createOn(account)
}
```

Setting up Account Entities

- Tests are short again! Now we're cooking!
  - Anything a user omits, we assume defaults for

```
val service: Service = builder.
    service(Service.Generic).
    escalationPolicy(epBuilder).
    createOn(account)
```

- The builder is a description of the entity.
  - `createOn` reifies the entity and all of its dependent resources

Setting up Account Entities

```
trait ResourceBuilder[T <: Resource] {
  def createOn(account: Account)(implicit subdomainClient: SubdomainAPI): T
}

case class ServiceBuilder(
  serviceType: Service.Type,
  escalationPolicyBuilder: Option[ResourceBuilder[EscalationPolicy]] = None,
  acknowledgementTimeout: Option[Duration] = None,
  autoResolveTimeout: Option[Duration] = None) extends ResourceBuilder[Service] {

  protected type This = ServiceBuilder

  def autoResolveTimeout(timeout: Duration): This =
    copy(autoResolveTimeout = Some(timeout))

  def acknowledgementTimeout(timeout: Duration): This =
    copy(acknowledgementTimeout = Some(timeout))

  def escalationPolicy(epBuilder: ResourceBuilder[EscalationPolicy]): This =
    copy(escalationPolicyBuilder = Some(epBuilder))

  def escalationPolicy(ep: EscalationPolicy): This =
    copy(escalationPolicyBuilder = Some(ConstantResourceBuilder(ep)))

  override protected def createOn(account: Account)
    (implicit subdomainClient: SubdomainAPI) = ??? // we'll get to this
}
```

Setting up Account Entities: Load Considerations

As with account provisioning, with entities we can either:

1. Create new entities each test run
2. (Somehow) re-use previously created entities

100s of tests each run every 5 minutes

=> 28K+ services, escalation policies, users, created per day

Reusing Account Entities

- PagerDuty's API allows querying for entities with a given name. e.g.,
  `GET https://<account>.pagerduty.com/api/v1/services?query=Database`
- Returns all services with "Database" in their name (`service.name like "%Database%"`)
- This is lookup! We can build a cache on this! What should the key be?
- Use the hashCode of the builder

Reusing Account Entities

```
val b: ServiceBuilder = builder.service(Service.Generic).
  autoResolveTimeout(45.minutes).
  acknowledgementTimeout(15.minutes)
```

Equivalent to

```
val b = ServiceBuilder(
  serviceType = Service.Generic,
  escalationPolicyBuilder = None,
  acknowledgementTimeout = 15.minutes,
  autoResolveTimeout: 45.minutes)
```

Builders are case classes.
Default hashCode implementation recursively accounts for hashCode of all members

```
b.hashCode == 74398412
```

Reusing Account Entities

```
trait ResourceBuilder[T <: Resource] {
  // The caching version!
  def getOrCreateOnAsync(account: Account)
    (implicit subdomainClient: SubdomainAPI): Future[T] = {
    val moniker = s"[${hashCode.toString}]"
    val existingResource = findExisting(account, moniker)(subdomainClient)
    existingResource rescue {
      case _: NoMatchingResourcesException =>
        asyncCreateOn(account, Some(moniker))
    }
  }

  // These methods overridden by builder subclass
  protected def asyncCreateOn
    (account: Account, requiredName: Option[String])
    (implicit subdomainClient: SubdomainAPI)
      : Future[T]

  def findExisting
    (account: Account, name: String)
    (implicit subdomainClient: SubdomainAPI): Future[T]
}
```

Reusing Account Entities: Hashing Gotchas

- Originally had

```
val moniker = hashCode.toString
```

- Now have

```
val moniker = s"[${hashCode.toString}]"
```

- Original has a subtle bug. See it?
- Remember that `LIKE` query?
- If b.hashCode == 743, we could get results with 743 *in* name (e.g., service.name == 984**743**103). Whoops.

Dealing with Issues

1. Account provisioning
2. Setting up account entities (e.g., services, EPs)
3. Waiting for occurrences (e.g., a new triggered incident)

Expecting Occurrences

- We started with:

```
// 4. Send the event to PagerDuty
triggerEvent.send

// 5. Wait for an incident
val incident = waitForIncidentToBeCreated // somehow?
```

- Synchronous short polling is sufficient

Expecting Occurrences

- As a test author, I want to concisely and declaratively specify the condition to wait for

```
/* Wait for a single triggered incident on an account */
val occurrence = incident on account where {
    incident: Incident => incident.state == Incident.Triggered
}
val incident: Incident = waitFor(occurrence)

/* Simultaneously wait for two incidents on a service  */
val occurrence1 = incident on service where {
  i: Incident => i.incidentNumber >= 5
}
val occurrence2 = incident on service where {
  i: Incident => i.incidentKey == "host0765"
}
val incident: Incident = waitFor(occurrence1 and occurrence2)
```

Expecting Occurrences

`waitFor` should transform to a short poll loop

```
val incident: Incident = waitFor(occurrence)
```

Should become

```
val pollingFrequency: Duration = ???
val maxTimeToWait: Duration = ???

val deadline = maxTimeToWait.fromNow
while (!occurrence.hasOccurred) {
if (Time.now > deadline)
  throw new TimeoutException(s"Did not complete within $maxTimeToWait")
 Time.sleep(pollingFrequency)
}
```

Nice to control timing parameters for `waitFor`

```
val incident: Incident = waitFor(occurrence).
  within(15.seconds). // maxTimeToWait
  checkEvery(3.seconds) // pollingFrequency
```

Expecting Occurrences

Other syntactic niceties

```
val occurrence = incident.
  on(service).
  withIncidentKey("abcd").
  withState(Incident.Triggered)
```

Expecting Occurrences

- How can we implement this DSL?

```
val occurrence = incident.on(account).withState(Incident.Triggered)
// Becomes while(!occurrence.hasOccurred)
val incident: Incident = waitFor(occurrence)
```

- Two entities:
  - Occurrences
  - Expectations
- Occurrences return entities
- Expectations wait for occurrences
- Performing conjunction between occurrences is a little hairy

Expecting Occurrences

```
trait Occurrence {
  def hasOccurred: Boolean
}

trait Occurrence1[T] extends Occurrence {
  protected type Predicate = T => Boolean

  def apply(): T
  def and[T2](other: Occurrence1[T2]) =
    new Occurrence2[T, T2](this, other)
}

sealed class Occurrence2[T1, T2](occA: Occurrence1[T1], occB: Occurrence1[T2])
  extends Occurrence {
  def apply(): (T1, T2) = (occA.apply, occB.apply)
  def hasOccurred: Boolean = occA.hasOccurred && occB.hasOccurred
  def and[T3](other: Occurrence1[T3]) =
    new Occurrence3[T1, T2, T3](this, other)
}

// Analagous for Occurrence3[T1,T2,T3], Occurrence4[T1,T2,T3,T4]
```

Expecting Occurrences

- How do occurrences compose?

```
val occurrenceA: Occurrence1[Incident] = incident.on(service).where { ... }
val occurrenceB: Occurrence1[Incident] = incident.on(service).where { ... }
val occurrenceC: Occurrence1[Incident] = incident.on(service).where { ... }

val occurrence2: Occurrence2[Incident] = occurrenceA and occurrenceB
val occurrence3: Occurrence3[Incident] = occurrence2 and occurrenceC
// or
val occurrence3: Occurrence3[Incident] =
  occurrenceA and occurrenceB and occurrenceC

val incidents: (Incident, Incident, Incident) = occurrence3()
```

Expecting Occurrences

```
case class IncidentOccurrence(subdomainAPI: SubdomainAPI)
  extends Occurrence1[Incident] {
  type This = IncidentOccurrence

  var _predicate: Option[Predicate] = None
  var _account: Option[Account] = None
  var _service: Option[Service] = None
  var _states: Seq[Incident.State] = Seq()
  var _incidentKey: Option[String] = None

  var _incident: Incident = null

  // regular builder pattern stuff
  def on(service: Service): This = ...
  def on(account: Account): This = ...
  def where(predicate: Predicate): This = ...
  def withState(state: Incident.State): This = ...
  def withStates(states: Seq[Incident.State]): This = ...
  def withIncidentKey(incidentKey: String): This = ...
```

Expecting Occurrences

```
  def hasOccurred: Boolean = {
     if (_account == None)
       throw new IllegalArgumentException("account or service missing")

     val allIncidents = Await.result(subdomainAPI.incidents(/* Use instance vars */))
     val incidents =
       if (_predicate == None)
         allIncidents
       else
         allIncidents.filter(_predicate.get)

     if (!incidents.isEmpty)
       _incident = incidents.head

     !incidents.isEmpty
   }

   def apply: Incident = {
     _incident
   }
}

object IncidentOccurrence {
   def incident(implicit subdomainAPI: SubdomainAPI)
     = new IncidentOccurrence(subdomainAPI)
}
```

Expecting Occurrences

```
abstract sealed class Expectation(
    timeout: Duration,
    pollingFrequency: Duration,
    delayTime: Duration)
{
  type This

  def hasOccurred: Boolean

  protected def poll[Result](result: => Result): Result = {
    val deadline = timeout.fromNow
    if (delayTime > 0.seconds) Time.sleep(delayTime)

    while (!hasOccurred) {
      if (util.Time.now > deadline)
        throw new TimeoutException(s"Did not complete within $timeout")

      Time.sleep(pollingFrequency)
    }

    result
  }
}
```

Expecting Occurrences

```
sealed case class Expectation1[T](
    occurrence: Occurrence1[T],
    timeout: Duration = Expectation.DefaultTimeout,
    pollingFrequency: Duration = Expectation.DefaultPollingFrequency,
    delayTime: Duration = Expectation.DefaultDelayTime)
  extends Expectation(timeout, pollingFrequency, delayTime)
{
  type This = Expectation1[T]
  // inherits poll from base class
  def awaitResult: T = poll(occurrence())
  override def hasOccurred: Boolean = occurrence.hasOccurred
}
// analogous definition for Expectation2[T1, T2], Expectation3[T1, T2, T3]


object Expect {
  def apply[T](occurrence: Occurrence1[T]) = new Expectation1[T](occurrence)
}

// Usage
val occurrence = incident.on(service).withIncidentKey("abcd")
val incident: Incident = Expect(occurrence).awaitResult
```

Expecting Occurrences

- How to allow timing parameter control?

```
val incident: Incident = Expect(occurrence).
  within(15.seconds). // maxTimeToWait
  checkEvery(3.seconds) // pollingFrequency
```

Expecting Occurrences

```
abstract sealed class Expectation(
    timeout: Duration,
    pollingFrequency: Duration,
    delayTime: Duration)
{
  ...

  // builder pattern again!
  def within(time: Duration): This =
    copyProtected(time, pollingFrequency, delayTime)
  def checkEvery(period: Duration): This =
    copyProtected(timeout, period, delayTime)
  def waitAtLeast(minTime: Duration): This =
    copyProtected(timeout, pollingFrequency, minTime)

  protected type CopyFunction = (Duration, Duration, Duration) => This
  protected val copyProtected: CopyFunction
}

sealed case class Expectation1[T](...) {
  ...
  override protected val copyProtected: CopyFunction = copy(occurrence, _, _, _)
}
```

Dealing with Issues

1. Account provisioning
2. Setting up account entities (e.g., services, EPs)
3. Waiting for occurrences (e.g., a new triggered incident)

WatchDog in Production

- 100+ tests running continuously in production
- Different *rings* of tests
  - shorter tests run every 5 minutes
  - longer tests run every 30 mins

Issues Found by WatchDog

- Slow enqueuing of events (p99.9 > 20 seconds)
- LBs throwing 502s due to TLS issues between LBs and event enqueuer
- 500s due to a database node changeover and resizings
- Breaking API change (notification rules when creating contact method)
- Load balancer change causing bg work to be processed by only one worker
- Countless other issues caught before code went live

What else have we learned?

- Rare anomalies, exercised repeatedly, will fail (and page)
- Lots of expectations on client behaviour
- Tension between false failures and timing tolerances
- WatchDog tests the system
  Unstable system => unstable tests
- Majority of failures are transient system issues
- Lots of tests => lots of load

Thank you.

pagerduty.com