BANKING
TECHNOLOGY

4.x Training Overview

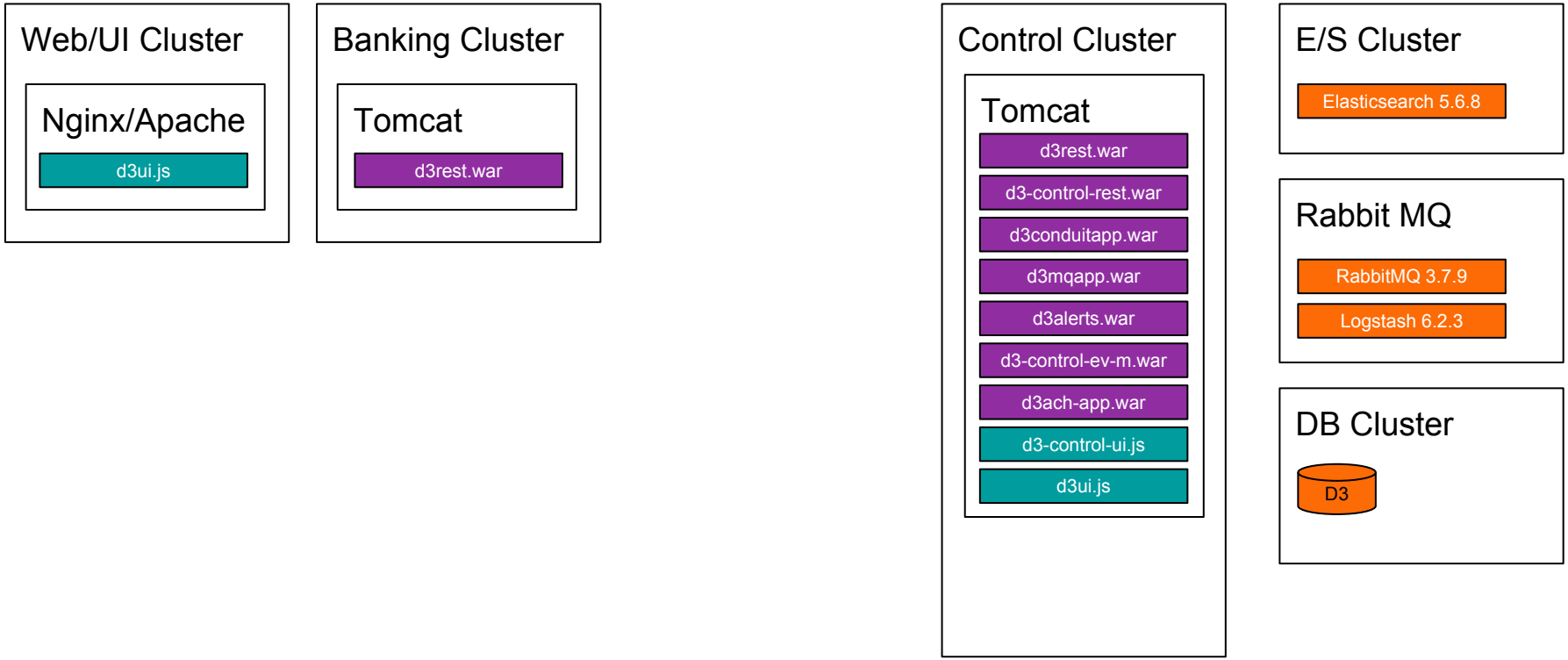
D3 Architecture

BANKING
TECHNOLOGY

4.x Training

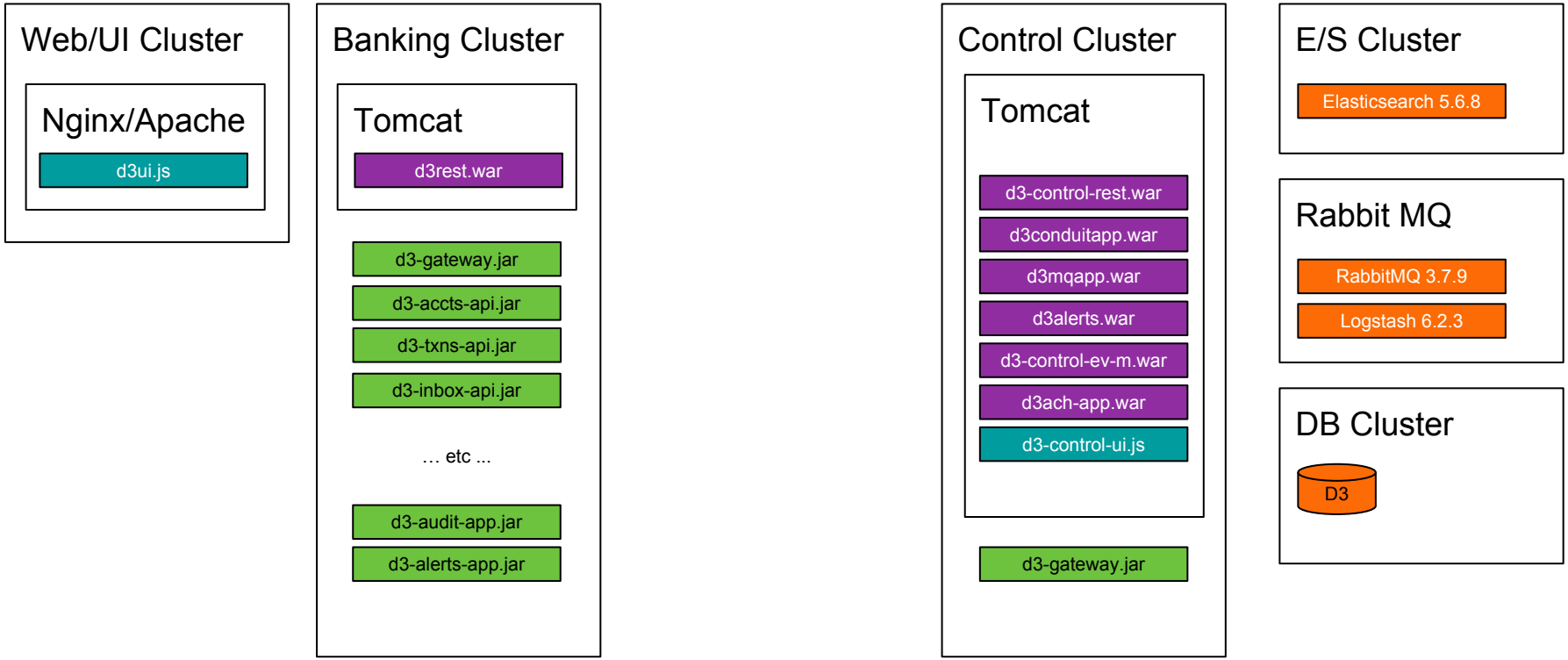
Introduction

Deployment 3.4



BANKING TECHNOLOGY

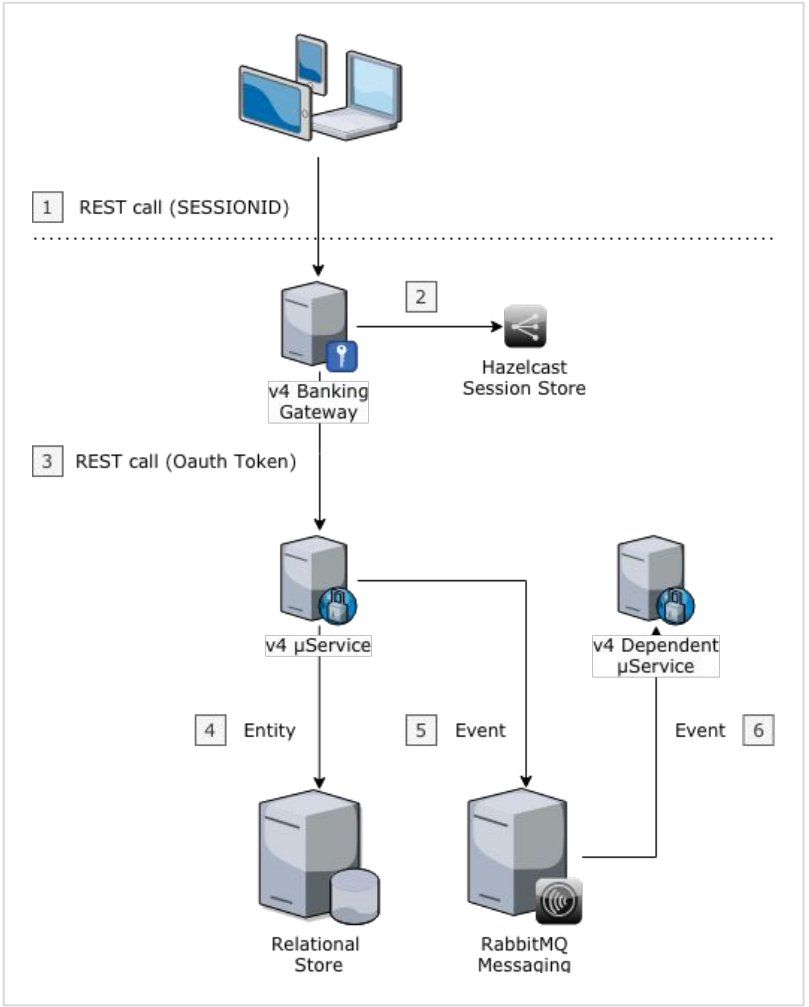
Deployment 4.x



BANKING TECHNOLOGY

Training Goals

- RESTful API
- Eventing API
- D3 Platform & SDK



D3 BANKING TECHNOLOGY

Agenda (Day 1)

- Architecture
  - RESTful API
  - Loose / No Coupling
  - Microservices
  - Event-Driven
  - Authentication Token
- D3 Platform Overview
  - Java 8+, Lombok, Swagger
  - Spring Framework, Spring Data JPA
  - Wiring (application.yml)
  - RabbitMQ
  - Apache Camel
- D3 Platform SDK
  - Context
  - Events
  - Exceptions
  - Validation

BANKING TECHNOLOGY

Agenda (Day 2)

D3 Core Banking Platform SDK

- Encryption / Masking
- Tenants
- Accounts (Consumer, Small Business)
- Users (Primary, Secondary, Shadow)
- Aspects
- Banking Gateway + Request Context

Hands-on Examples

- Banking API (Transaction Image Adapter)
- Event Listener (Simple Logger)

BANKING
TECHNOLOGY

4.x Training

Platform Overview

D3 BANKING TECHNOLOGY

Anatomy of a D3 Microservice

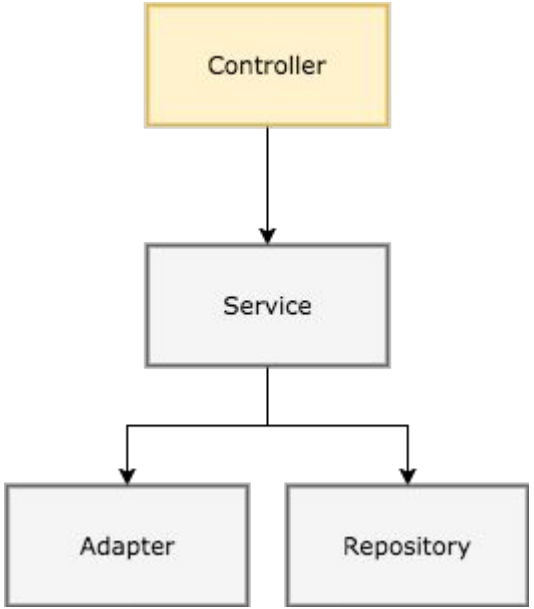
- 3-Tier architecture (Controller, Service, Repository)
- Pluggable adaptors for external integration
- Event-driven
- Architectural best-practices encoded in SDK via annotations and AOP
- API implemented using contract-first approach

Will review using a hypothetical enhancement request.

BANKING TECHNOLOGY

REST Endpoint Definition

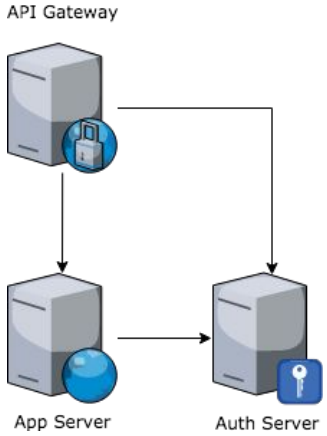
```
@RestController
class AtmController {

  @PostMapping
  @PreAuthorize("hasRole('ATM_ISSUE_CREATE')")
  @ApiOperation
  public AtmIssueApiDto reportIssue(
    @RequestHeader("device") String deviceUuid,
    @RequestBody @Valid AtmIssueApiDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

REST Endpoint Security

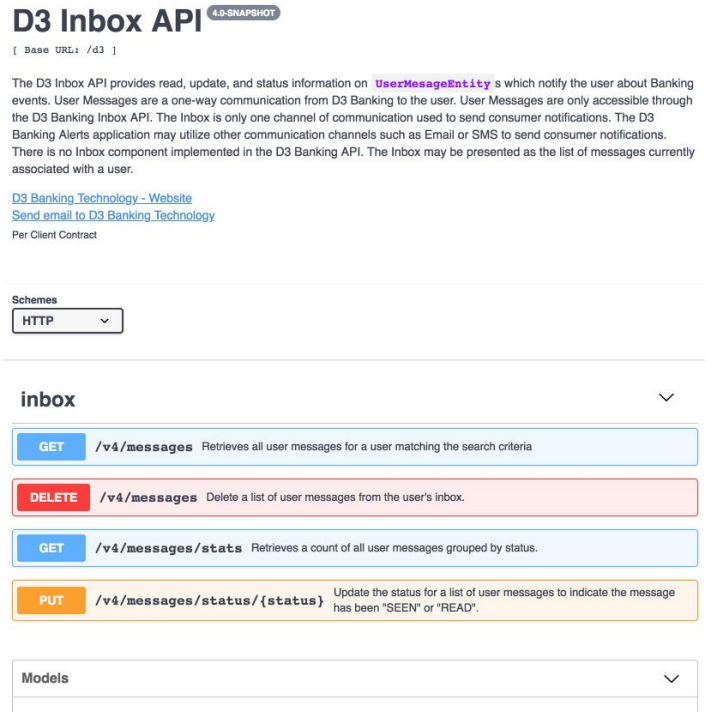
```
@RestController
class AtmController {

  @PostMapping
  @PreAuthorize("hasRole('ATM_ISSUE_CREATE')")
  @ApiOperation
  public AtmIssueApiDto reportIssue(
    @RequestHeader("device") String deviceUuid,
    @RequestBody @Valid AtmIssueApiDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

REST Endpoint Documentation

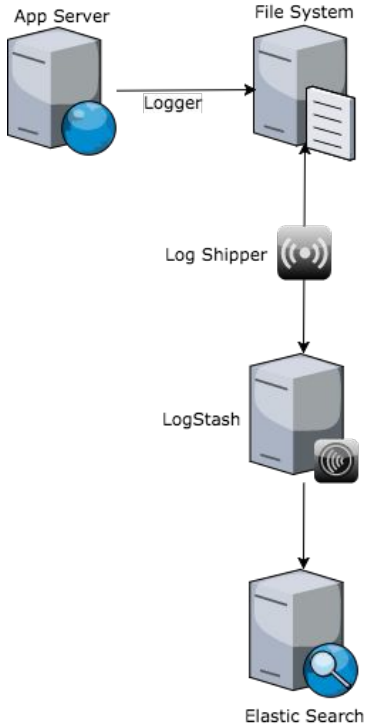
```
@RestController
class AtmController {

  @PostMapping
  @PreAuthorize("hasRole('ATM_ISSUE_CREATE')")
  @ApiOperation
  public AtmIssueApiDto reportIssue(
    @RequestHeader("device") String deviceUuid,
    @RequestBody @Valid AtmIssueApiDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

REST Endpoint Logging

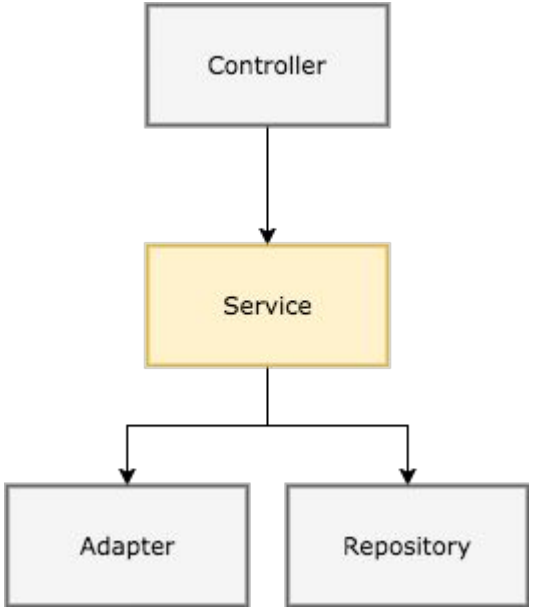
```
@RestController
class AtmController {

  @PostMapping
  @PreAuthorize("hasRole('ATM_ISSUE_CREATE')")
  @ApiOperation
  public AtmIssueApiDto reportIssue(
    @RequestHeader("device") String deviceUuid,
    @RequestBody @Valid AtmIssueApiDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

Service Definition

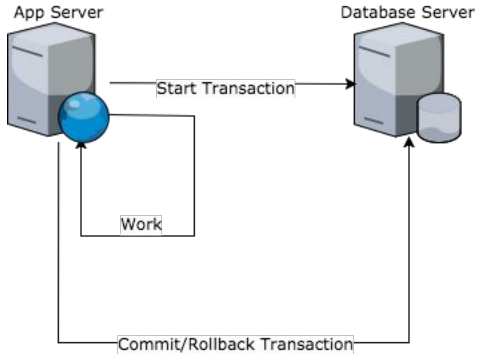
```
@Service
class AtmServiceImpl implements AtmService {

  @Transactional
  @BusinessEvent
  public AtmIssueDto createIssue(
    @Valid AtmIssueDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

Service Transactional Control

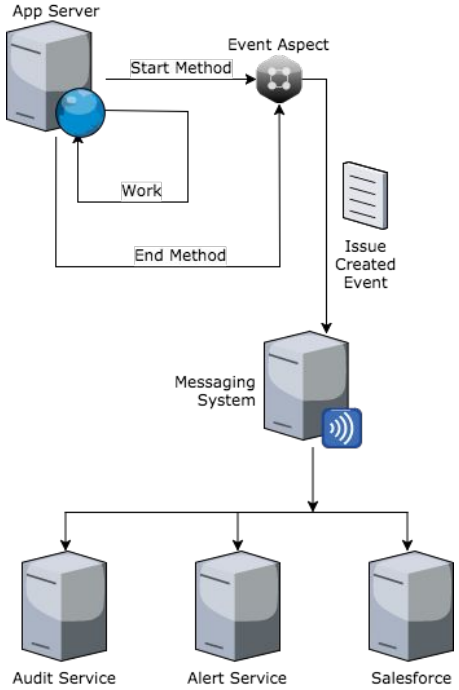
```
@Service
class AtmServiceImpl implements AtmService {

  @Transactional
  @BusinessEvent
  public AtmIssueDto createIssue(
    @Valid AtmIssueDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

Service Event Generation

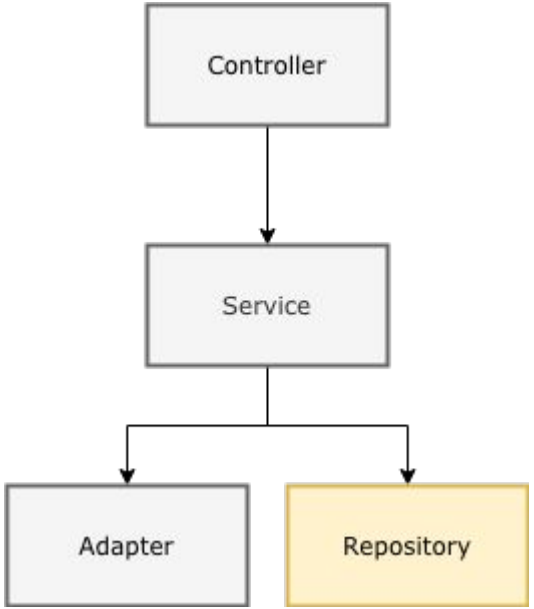
```
@Service
class AtmServiceImpl implements AtmService {

  @Transactional
  @BusinessEvent
  public AtmIssueDto createIssue(
    @Valid AtmIssueDto issue
  ) {
    ... etc ...
  }

}
```



BANKING TECHNOLOGY

Repository Definition

```
@Repository
interface AtmRepository extends Repository<> {


  List<AtmIssueEntity> findAllByUserId(
    @NotNull Long userId
  );

  AtmIssueEntity save(
    @Valid AtmIssueEntity issue
  );

}
```



BANKING TECHNOLOGY

Repository Definition

```
@Repository
interface AtmRepository extends Repository<Integer, AtmIssueEntity> {


  List<AtmIssueEntity> findAllByUserId(
    @NotNull Long userId
  );

  AtmIssueEntity save(
    @Valid AtmIssueEntity issue
  );

}
```

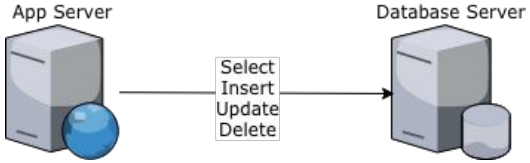
BANKING TECHNOLOGY

Repository Definition

```
@Repository
interface AtmRepository ex                    ory<> {

  List<AtmIssueEntity> findAllByUserId(
    @NotNull Long userId
  );

  AtmIssueEntity save(
    @Valid AtmIssueEntity issue
  );

}
```

select *
from atm_issue
where user_id = ?

insert / update



BANKING
TECHNOLOGY

4.x Training

application.yml

BANKING TECHNOLOGY

Server Config

```
server:
 port: 8500
 tomcat.accesslog:
   directory: ${logging.path}
   enabled: true
logging:
 path: ${user.home}/d3/logs/${spring.application.name}
 file: ${logging.path}/application.log
 level:
   com.d3banking.audit: DEBUG
```

access log (tomcat)

application log (spring)

BANKING TECHNOLOGY

Application Config

```
spring:
 application:
   name: d3-audit-subscriber
 main.banner-mode: "off"
 mvc:
   favicon.enabled: false
   formcontent.putfilter.enabled: false
```

yaml, so whitespace is important

BANKING TECHNOLOGY

Datasource Config

```yaml
spring:
 datasource:
   url: jdbc:mysql://localhost:3306/d3?useSSL=false
   username: d3
   password: d3
   hikari:
     maximum-pool-size: 5
     pool-name: d3-connection-pool
     connection-timeout: 5000
     transaction-isolation: TRANSACTION_READ_COMMITTED
```

RabbitMQ Client Config

```
rabbitmq:
  addresses: localhost:5672
  username: d3banking
  password: d3banking
```

Oauth2 Client Config

```
security.oauth2:
 resource:
   jwt:
     key-value:
       -----BEGIN PUBLIC KEY-----
       … TRIMMED ...
       -----END PUBLIC KEY-----
 user:
   name: client
   password: client
```

4.x Training

RabbitMQ

RabbitMQ: AMQP Overview

Java Messaging System (JMS)

Three primary abstractions:

- Message
- Queue or Topic

Publisher is in charge. Publisher must decide at design time which model is used, and what the details are (persistence, TTL, etc).

Advanced Message Queuing Protocol (AMQP)

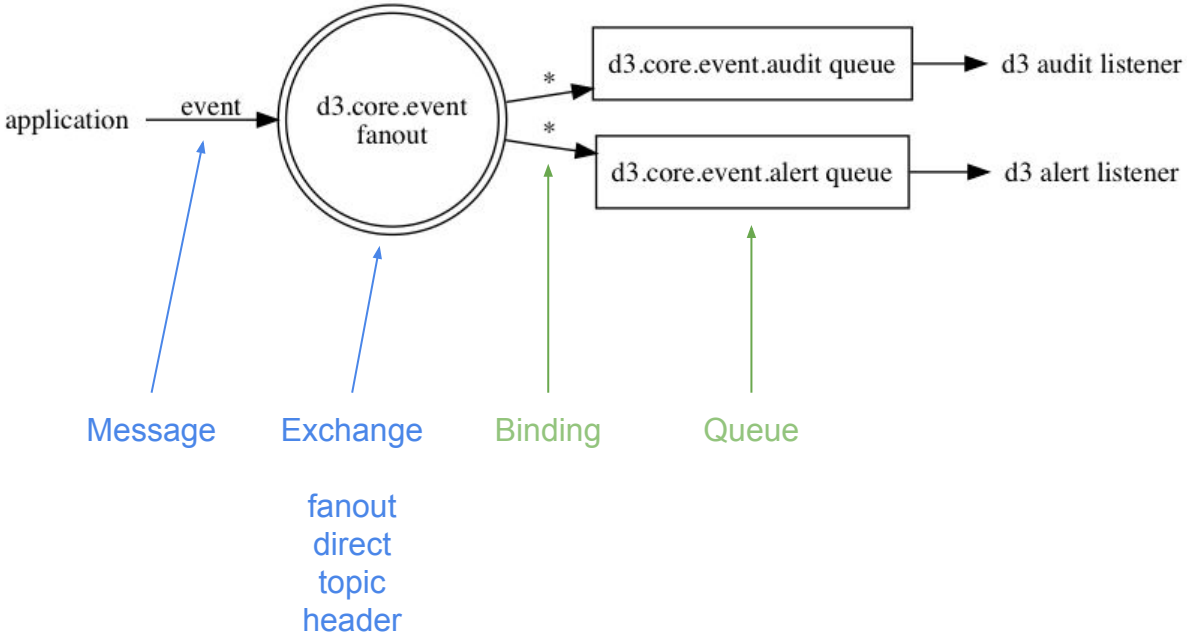
Five primary abstractions:

- Message
- Exchange
- Queue
- Routing Key
- Binding

Consumer is in charge. Publisher only cares about the Exchange. Consumer(s) create their own queues and bind them to the Exchange based on their needs (routing key(s), persistence, TTL, etc).

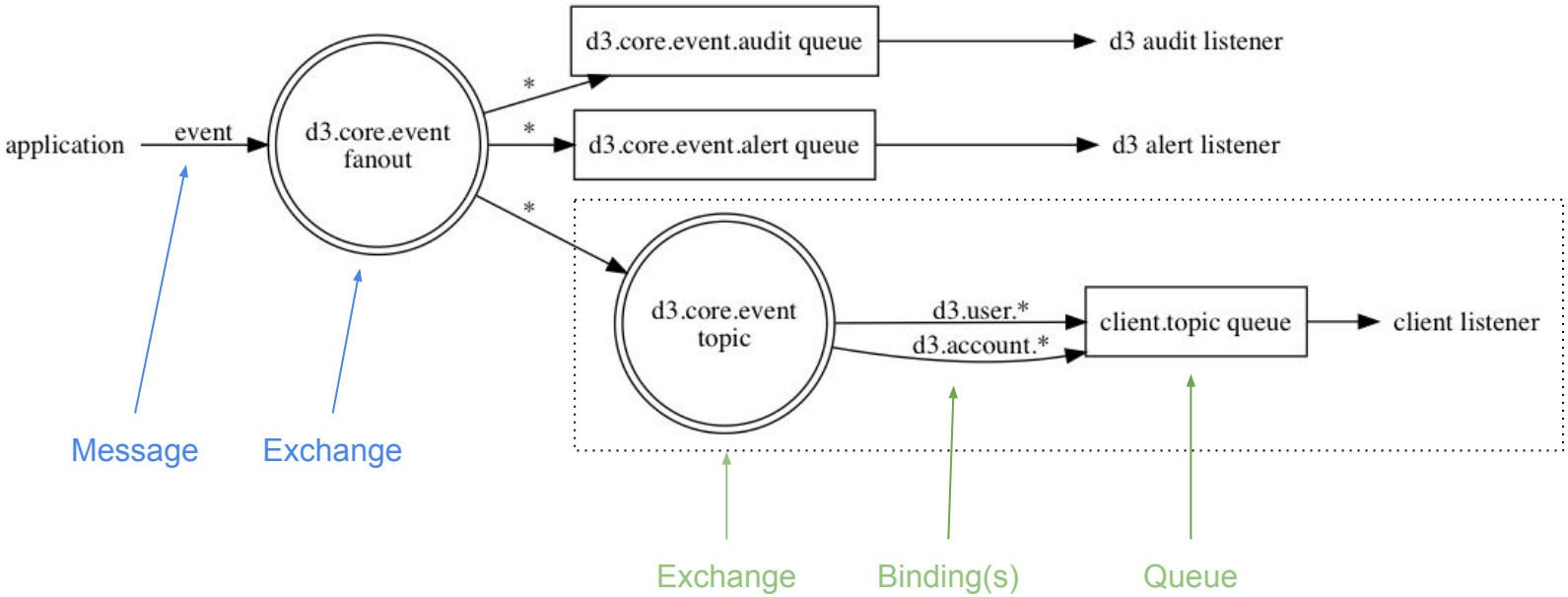
BANKING TECHNOLOGY

RabbitMQ: D3 Core Events OOTB



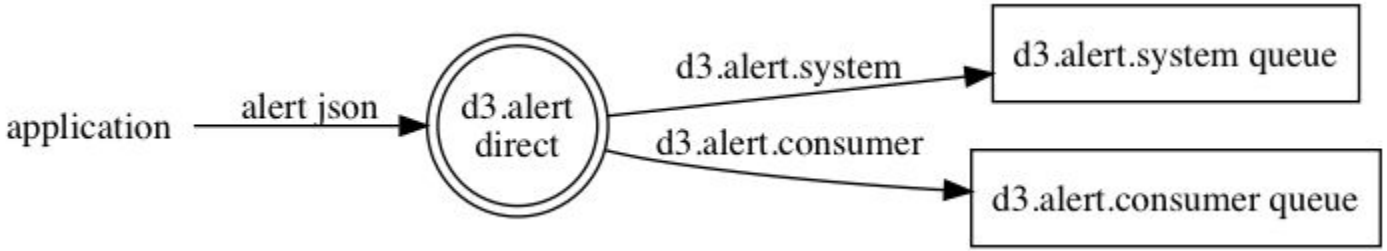
BANKING TECHNOLOGY

RabbitMQ: D3 Core Events Extended



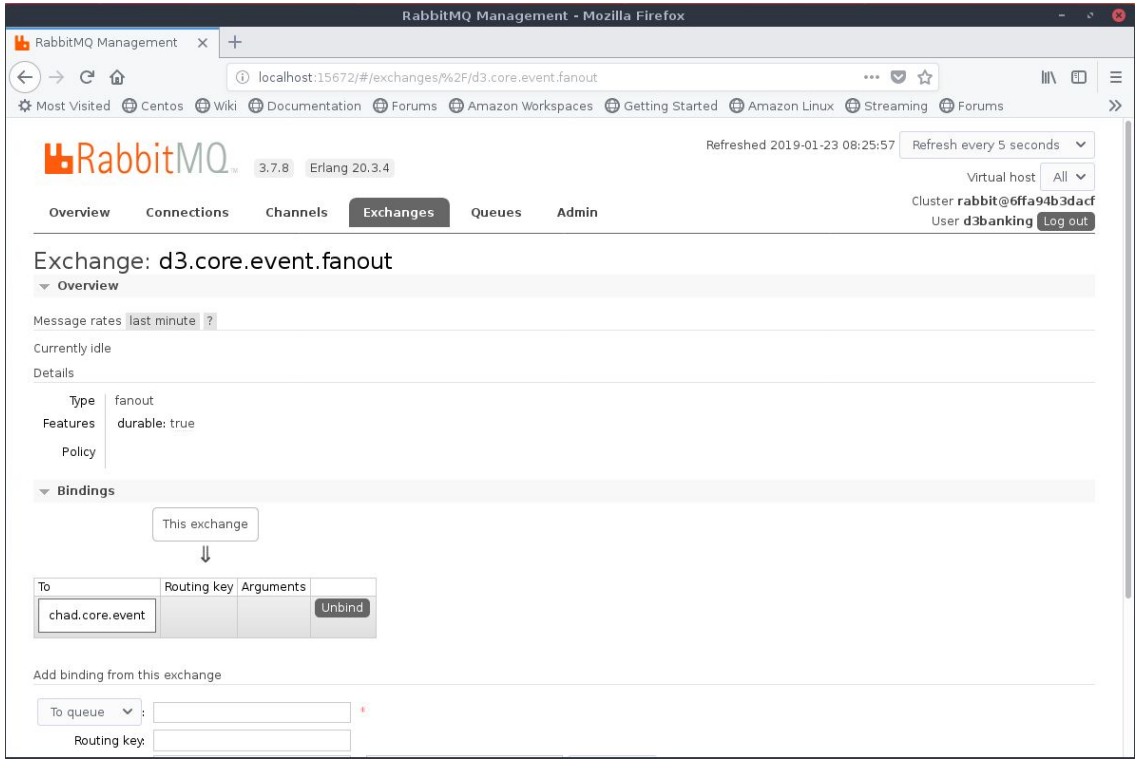
RabbitMQ: D3 Alerts

In addition to formalizing and publishing the D3 Event Catalog, 4.0 will also begin to expose non-RESTful API for those parts of the system where its meaningful to do so….



BANKING TECHNOLOGY

RabbitMQ: Hands-on



BANKING
TECHNOLOGY

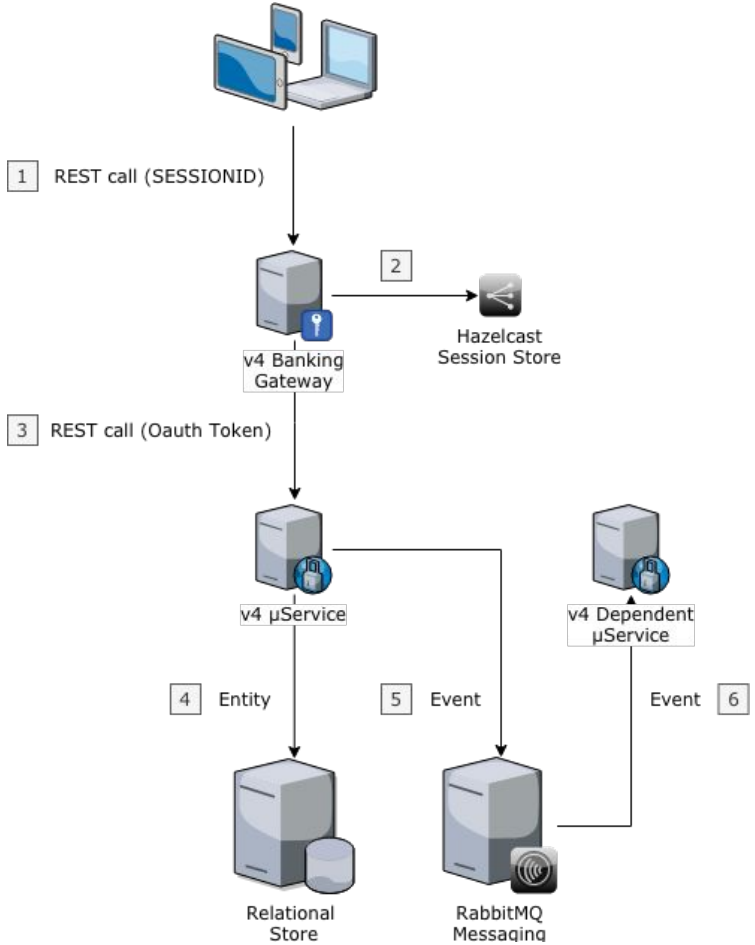
4.x Training

Platform SDK

BANKING TECHNOLOGY

Request Lifecycle

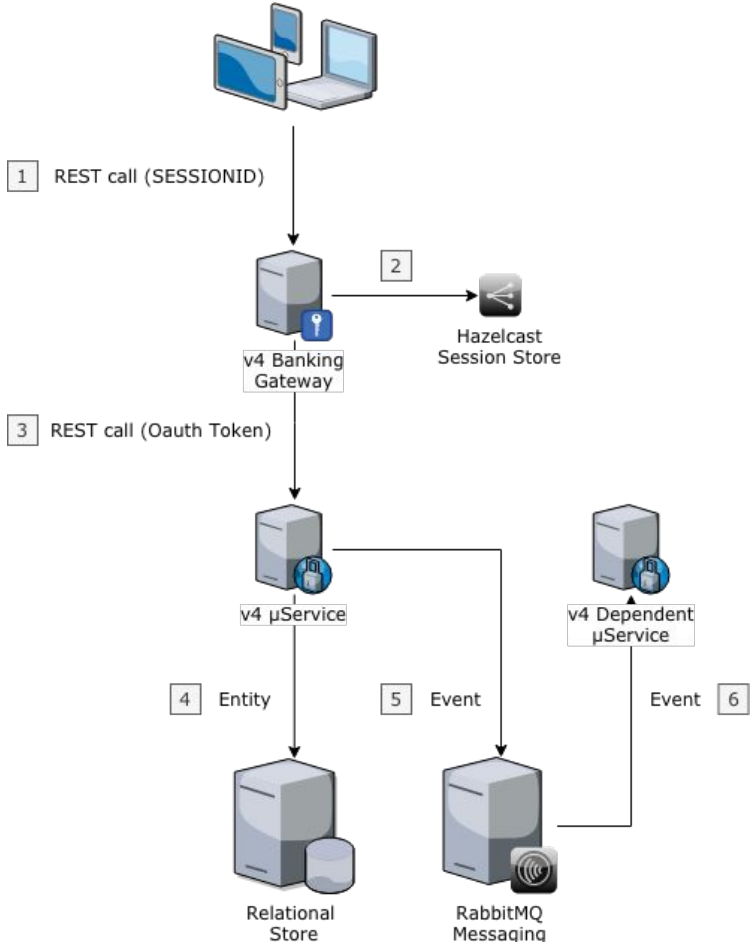
1. An HTTP-based RESTful request comes into the D3 Banking API Gateway from a Web or Mobile application-based `Client`.
2. The Gateway, which acts as an Oauth2 token relay, looks up the Oauth2 token based on the `SESSIONID`.
3. If an Oauth2 token exists and is still valid, the Gateway forwards the request to one of the `Server` instances running the microservice that has been registered for the given request URI. Otherwise, if no Oauth2 token could be resolved for the request, then the Gateway will reject the request with the appropriate HTTP status code (`401`).



BANKING TECHNOLOGY

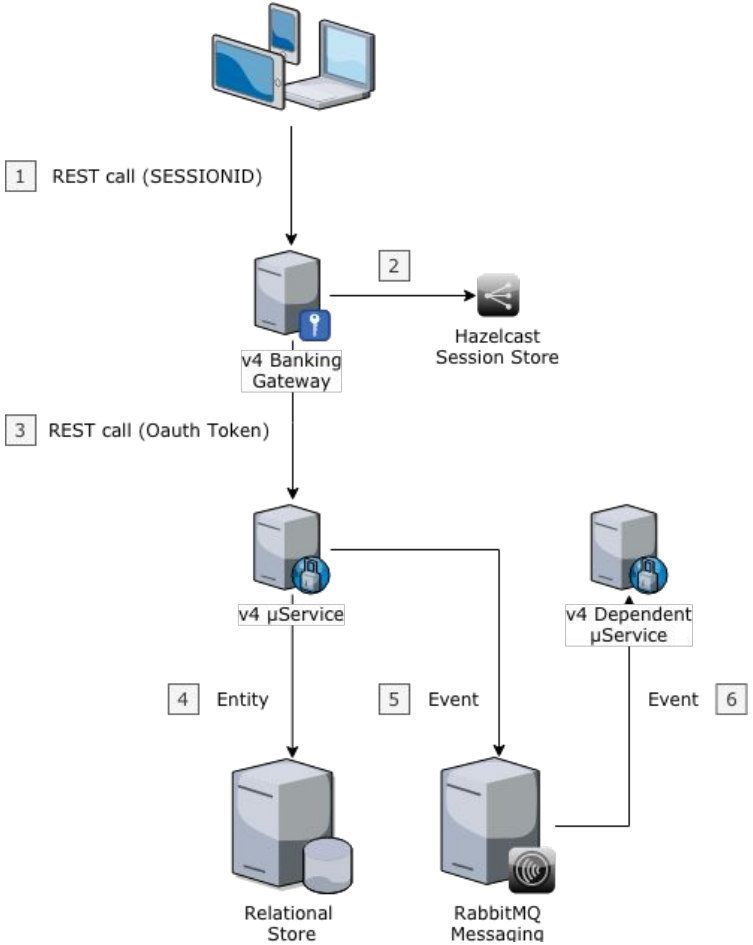
Request Lifecycle

4. The microservice processes the request using the following steps:
    i. It decodes the Oauth2 token to ascertain the identity of the `Actor` making the request. Since D3 is a multi-tenant system, it will also contain the identity of the `Tenant` that owns the data that is touched by the API request (including the `Actor`). If the decoding fails, then the microservice will reject the request with the appropriate HTTP status code (`401`).
    ii. It validates the request in terms of syntactic and semantic correctness (usually via a Spring `Controller`). If the validation fails, then the microservice will reject the request with the appropriate HTTP status code (`400`).
    iii. It processes the request (usually in a transaction via a Spring `Service`) which results in an `Entity` being persisted to the relational store. Any failure at this point would most likely result in the request failing with the appropriate HTTP status code (`500`).



Request Lifecycle

5. The microservice generates an `Event` to record the `Action` that took place in step 4, and broadcasts it via RabbitMQ. If the request failed, it will still generate the `Event`, but with a status indicating that the request `FAILED`.
6. One or more downstream service(s) are notified asynchronously of the `Event` that was broadcast in step 5.



Domain Models

The following domain models have been encoded in the D3 SDK since they form an important part of the activity documented above:

- `Actor` represents the person or process that is initiating some activity
- `Tenant` represents the entity that owns the data affected by the activity
- `Client` represents the device used by the actor to perform the activity
- `Server` represents the hardware on which the activity was processed

These domain models are implemented as Plain Old Java Objects (POJOs) and represent a "snapshot" of the activity described above, and form the backbone of the eventing and auditing of said activity. They are immutable once constructed.

BANKING TECHNOLOGY

Domain Models: Actor

An `Actor` in the D3 platform represents the person or process associated with an activity (e.g., API request, background process, maintenance script, etc).

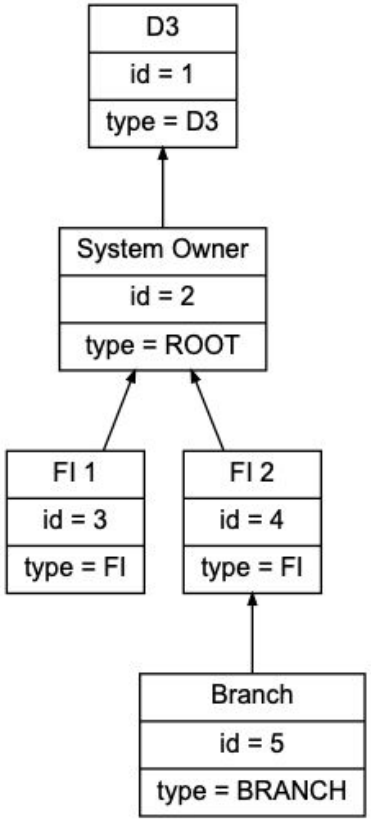
- `type` is a non-null attribute that contains the type of user or process that has initiated the activity. One of `ANONYMOUS`, `CONSUMER`, `INTERNAL`, `D3PROCESS`, or `D3SCRIPT`.
- `id` is an optional attribute that contains the id of the authenticated user or process that the activity is associated with.
- `actingAsId` is an optional value that contains the id of the consumer that the user or process is acting on the behalf of.

Domain Models: Tenant

The multi-tenant features provided by D3 are based on the concept of a Company Hierarchy.

There are 2 special companies in the diagram, `D3` and `System Owner`. These 2 companies are automatically seeded by D3 when the system is first installed. These 2 companies are special, and are not considered Tenants. However, each of the companies in the hierarchy *below* the `System Owner` are considered by D3 as Tenants of the system owned and operated by the `System Owner`.

All data captured by D3 can be tied back to exactly one `Tenant`.



Domain Models: Client

The `Client` is used to capture the details of the API client that initiated the activity. If the activity was initiated by a `D3PROCESS` or `D3SCRIPT`, then the `Client` will not be applicable.

The `Client`, once created, is immutable and composed of the following fields:

- `ipAddr` contains the IP address of the client.
- `userAgent` contains the user agent (e.g., browser) of the client.
- `channel` contains the channel being used by the client (e.g., `WEB`, `MOBILE`).
- `deviceId` contains the ID of the device being used (only applicable when the channel is `MOBILE`).
- `requestId` contains the UUID of the HTTP request.
- `sessionId` contains a unique (tracking) ID of the authenticated session. Should not match the actual session ID.

Domain Models: Server

The `Server` is used to capture the details of the server that was used to process the given activity. The `Server`, once created, is immutable and composed of the following fields:

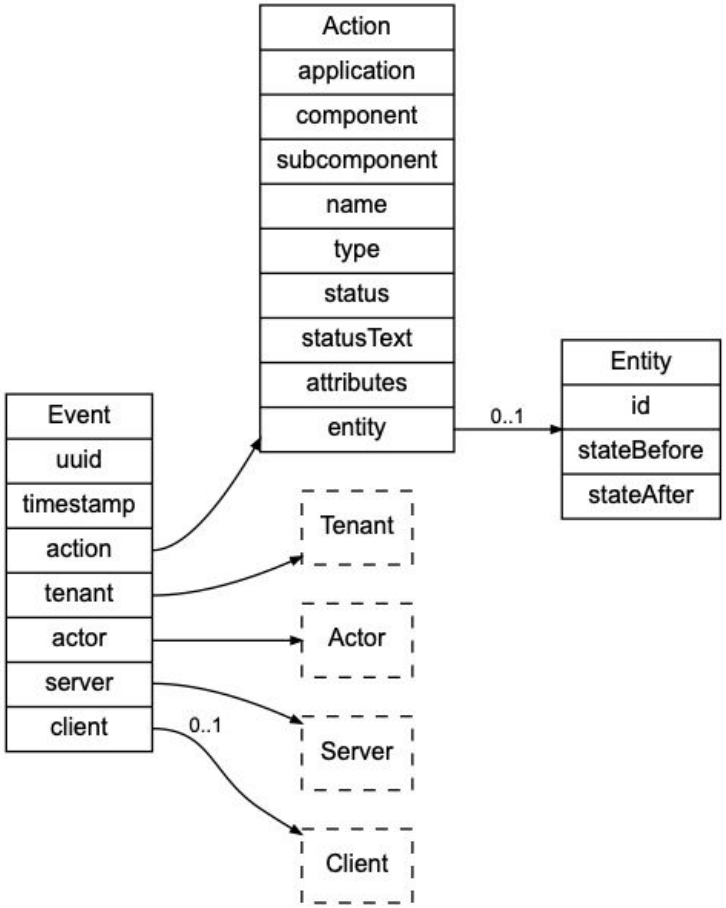
- `ipAddr` contains the IP address of the server.
- `hostName` contains the host name of the server.

BANKING TECHNOLOGY

Domain Models: Event

The D3 `Event` object encodes what D3 considers to be the best practices for *what* data should be captured for every event:

- `uuid` contains the unique identifier for the event.
- `timestamp` contains the date & time this event was generated.
- `action` contains the `Action` that was performed that triggered the event.
- `actor` contains the `Actor` that initiated the event.
- `tenant` contains the `Tenant` that owns the data associated with the event.
- `server` contains the `Server` that processed the event.
- `client` contains the `Client` that was used by the `Actor` to initiate the event.



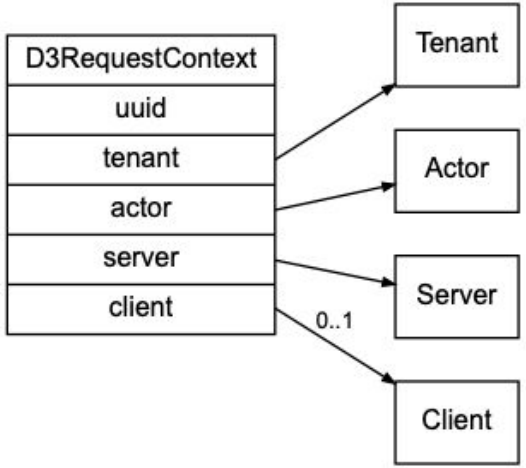
BANKING TECHNOLOGY

Request Context

Each D3 microservice needs to accommodate multi-tenancy, consumer users *and* small business users, users acting on the behalf of other users, desktop and mobile application-based sessions, etc. The purpose of the `D3RequestContext` is to encapsulate this "context" information into a single place.

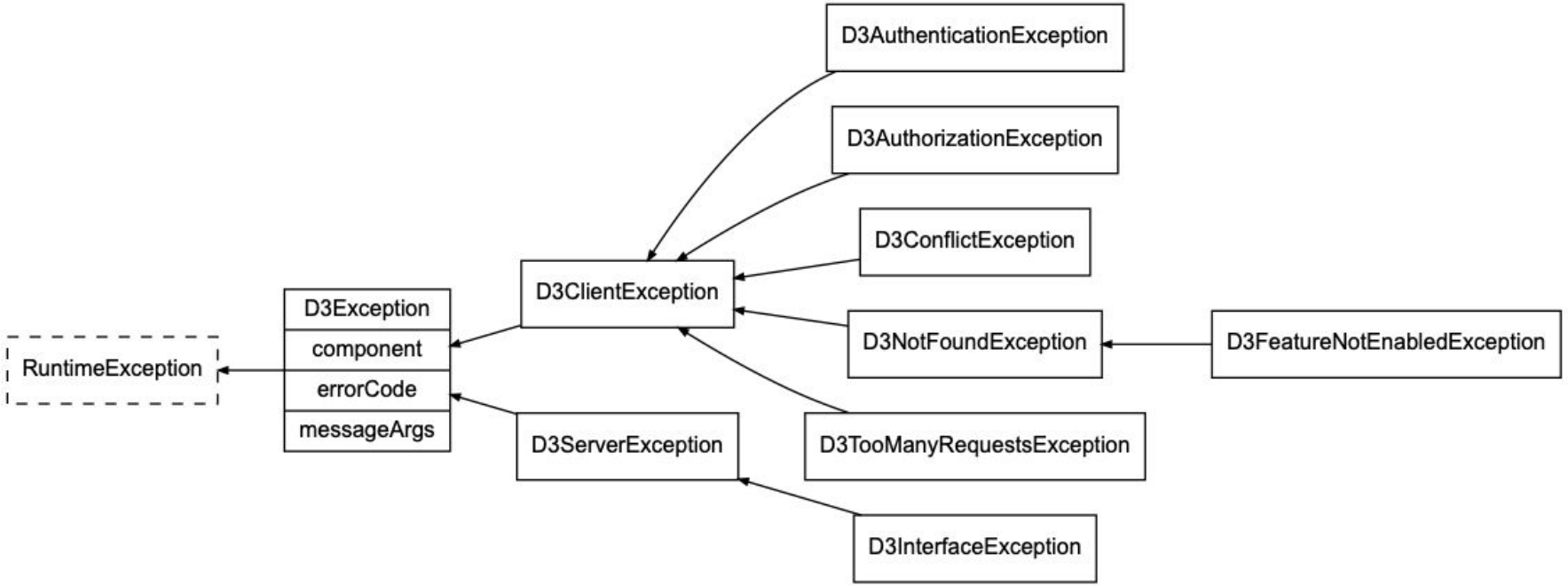
How the `D3 Request Context` gets populated is dependent on the scenario:

- For an API call, it is most likely populated via an interceptor.
- For a background process / job, it is most likely populated by the Job itself at the beginning of each run.



BANKING TECHNOLOGY

D3 Exception Hierarchy

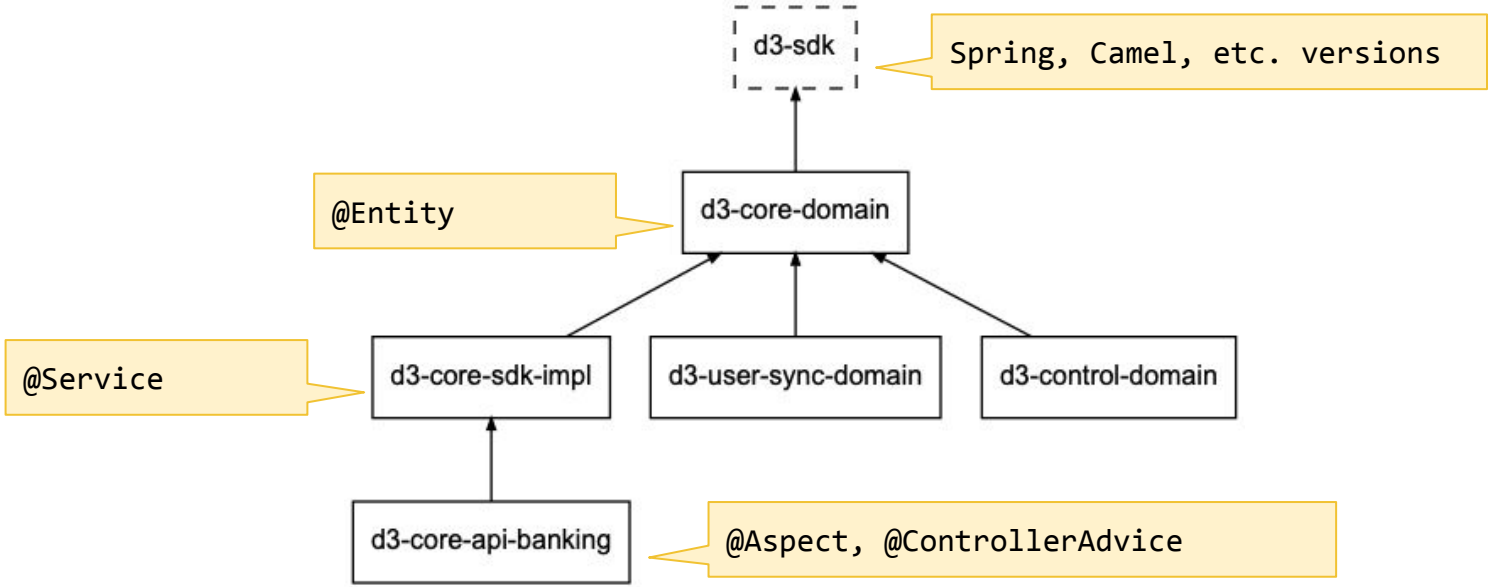


BANKING
TECHNOLOGY

4.x Training

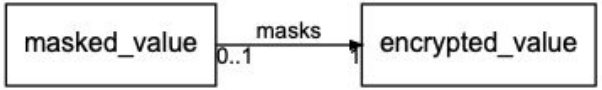
Core SDK

Core SDK Modules



Automatically configured via `@ComponentScan("com.d3banking.config")`

Encryption & Masking



One of the few core services that exposes the `@Entity` rather than a DTO due to the fact that the entities managed are owned by other entities (e.g., `Account` and `UserProfile`).

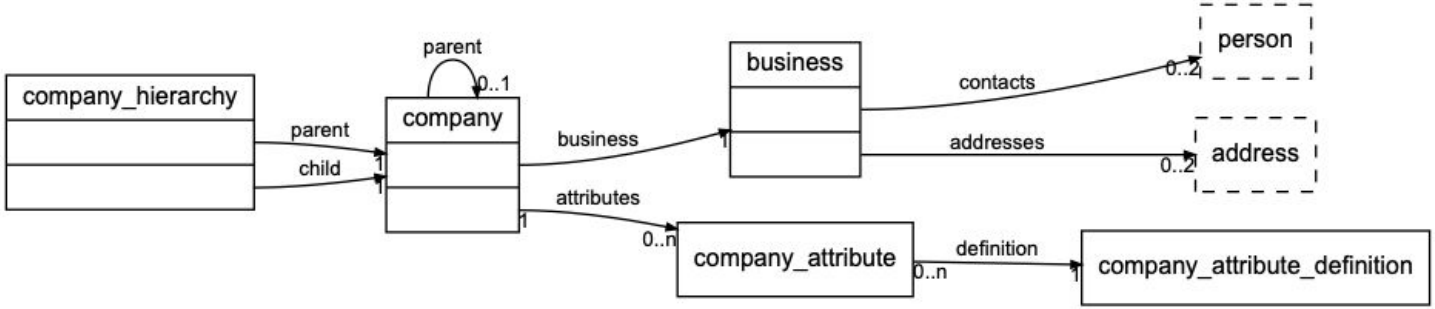
Functionality:

- Decryption
- Encryption
- Masking

Encryption / Decryption based on selected adapter. Current out of box is `ClearEncryptionAdapter`.

BANKING TECHNOLOGY

Multi-Tenancy via Company Service



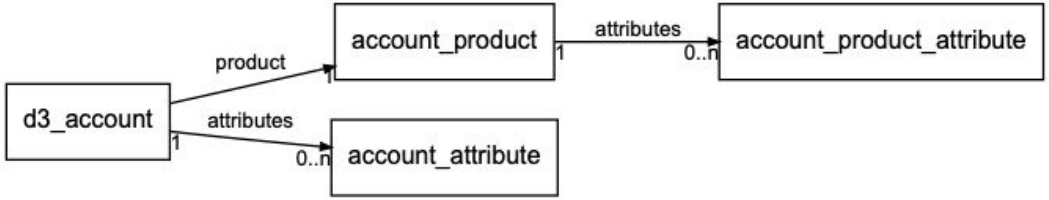
Functionality:

- Get Tenant
- Get Attribute(s) for Tenant
- Get Adapter for Tenant

BANKING TECHNOLOGY

Account Service

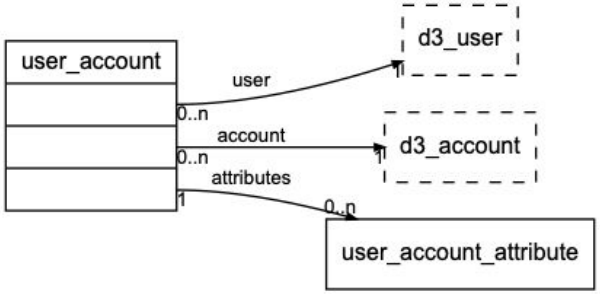
Usually not used directly since most queries are in the context of a User Account.



Functionality:

- Get Account Product
- Get Account
- Get Unmasked Account Number

User Account Service



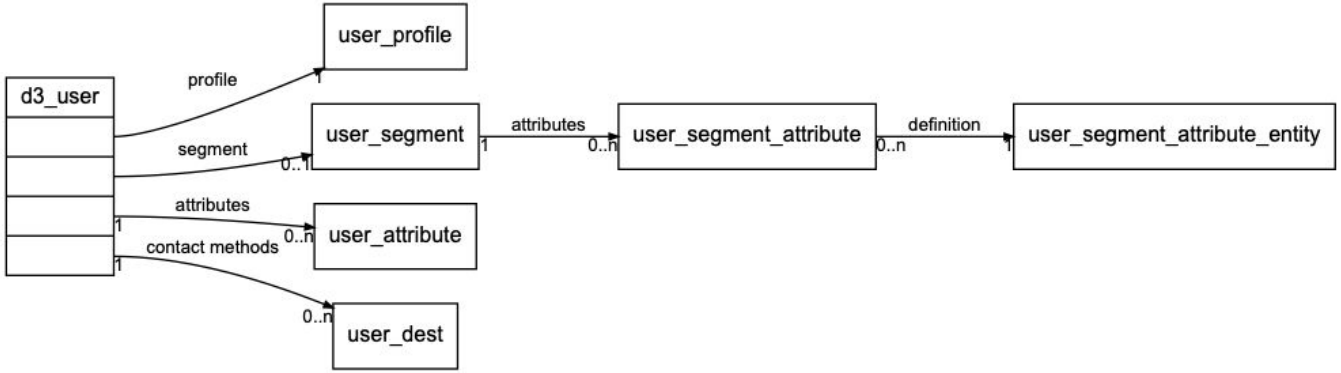
Will typically use this instead of Account Service.

Functionality:

- List User Accounts for User
  Optimized for speed by returning Summary DTO (no attributes). Requires 1 I/O.
- Get User Account Detail
  Returns full DTO (User Account -> Account -> Product) and attributes at each level. Requires 4 I/Os.
- Get Preferred Balance
- Get Unmasked Account Number

BANKING TECHNOLOGY

User Service



Functionality:

- Get User
- Get / Set User Attribute
- Get User Segment Attribute

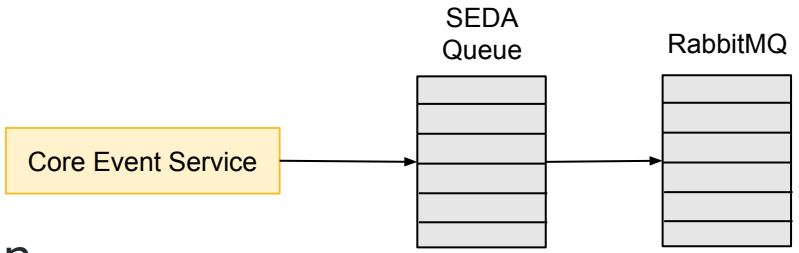
BANKING TECHNOLOGY

Event Service

Used to manually broadcast an `Event`. Uses Camel's SEDA queue to allow for non-blocking publish to Rabbit MQ with retry-forever behavior built-in.

Only 1 method:

- `void broadcast (@Valid Event)`



BANKING
TECHNOLOGY

4.x Training

Core Banking API

Banking Request Context Interceptor

Populates the D3 Request Context:

- Actor
- Tenant
- Client
- Server

Also uses Gateway's headers to populate:

- Request ID
- Session (tracking) ID

The latter 2 allow tracking across gateway and microservice(s).

BANKING TECHNOLOGY

Banking Api Event Aspect

Supplies the behavior for the `@BusinessEvent`:

- AOP on @Service
- Introspection
  - `@BusinessEvent.CaptureState`
  - `@BusinessEvent.CaptureId`
- `D3RequestContext` for context
- `CoreEventService` for broadcast

D3 BANKING TECHNOLOGY

Banking Exception Handler

Registers a global exception handler to properly format an error response:

- Localization via `L10nService`
- HTTP Status code based on `D3Exception` hierarchy