XCONF

Moving away from null and exceptions: An alternative way of error handling

Mario Fernandez
Andrei Bechet

ThoughtWorks®

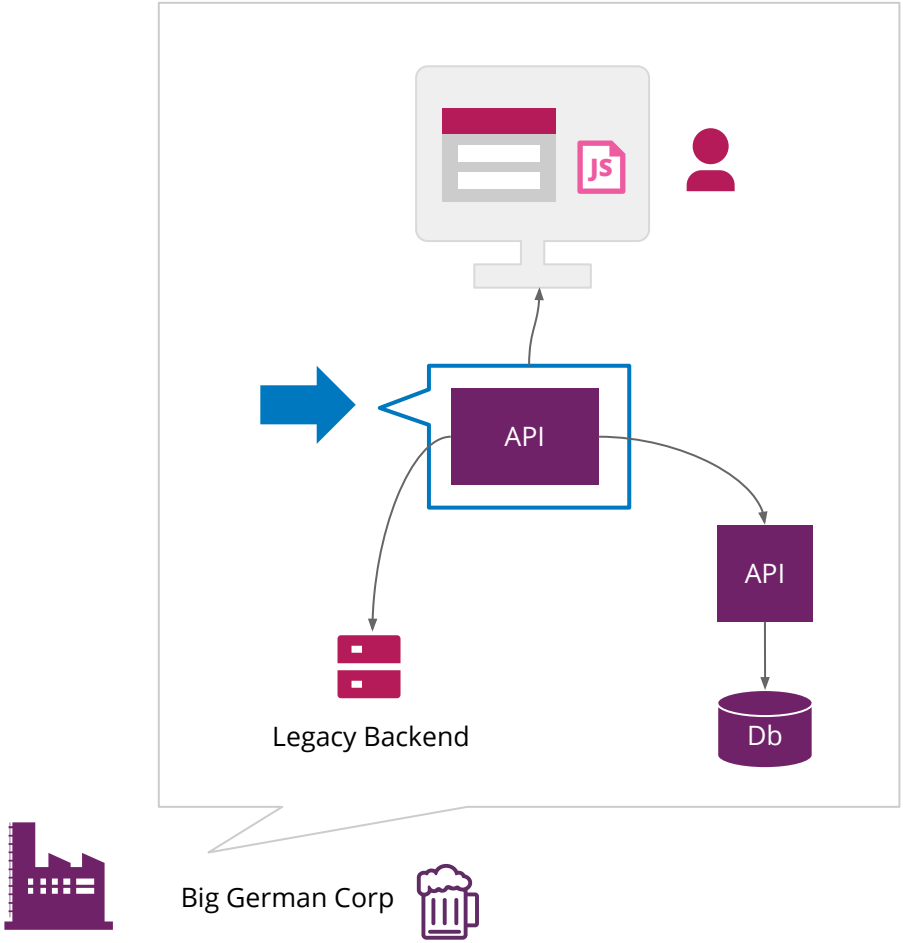
What to expect

The two takeaways

Let’s stop **using null**

Let’s stop **abusing** **exceptions**

Some Context

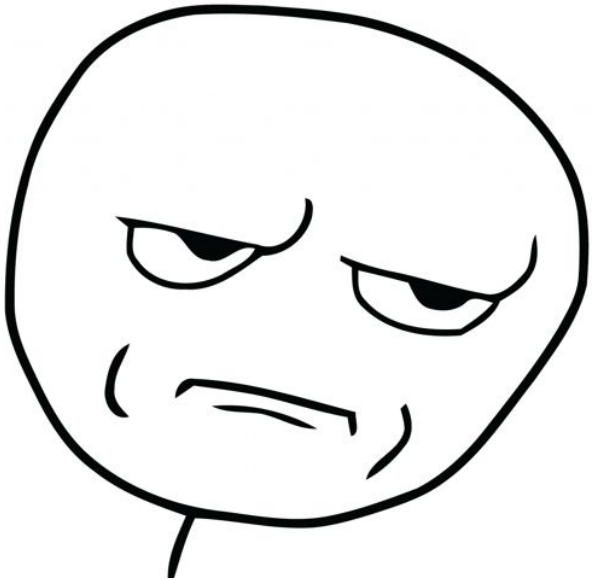

JS
API
API
Db
Legacy Backend
Big German Corp


Kotlin


ARROW

Null

Did you have a **NullPointerException** lately in a production system?



```
public static boolean isAdmin(List<Scope> scopes) {
    if(scopes == null) {
      return false;
    }

    Scope adminScope = findAdminScope(scopes);

    if(adminScope == null) {
      return false;
    }

    return adminScope.isValid();
}
```

Ad-hoc Error Handling and it pollutes

Null values sidestep the type system

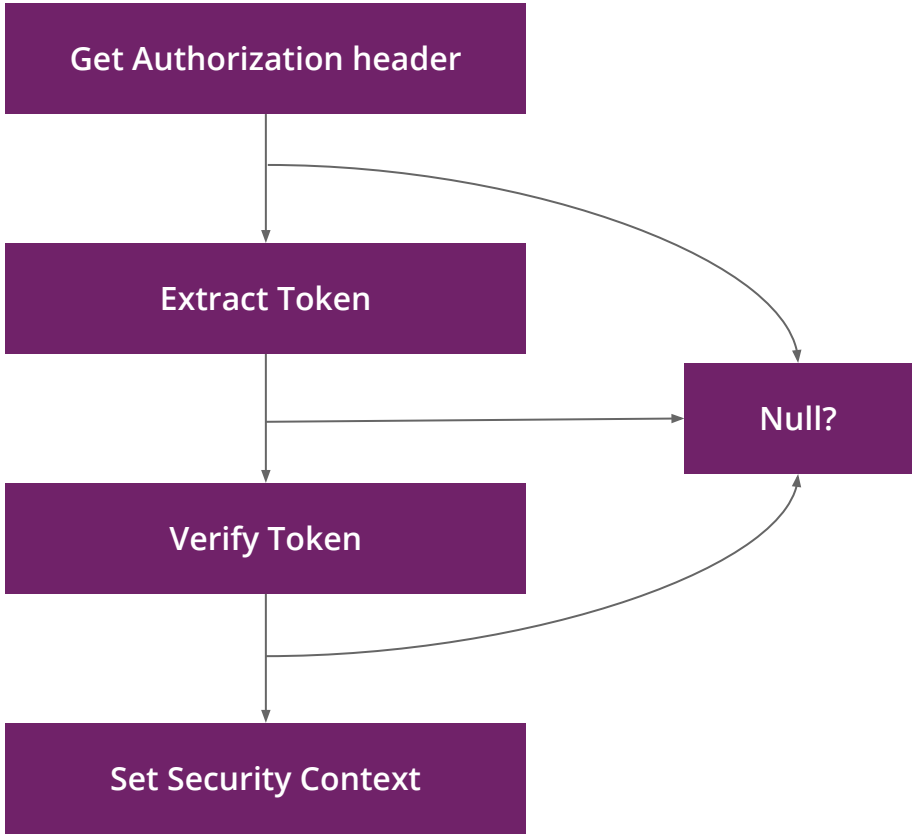
Runtime instead of compile time feedback

Nullable Types

```
Authorization: Bearer bGciOi...JIUzI1NiIs
```

```
fun String.extractToken(): String? = if (startsWith("Bearer"))
    split(" ").last()
else
    null
```

```
header.extractToken()
  ?.let { token -> doStuff(token) }
```


Get Authorization header
Extract Token
Null?
Verify Token
Set Security Context

```
request.getHeader(Headers.AUTHORIZATION)
  ?.let { header ->
    header.extractToken()
      ?.let { jwt ->
        verifier.verify(jwt)
          ?.let { token ->
            SecurityContextHolder.getContext().authentication = token
          }
      }
  }
```

Data Types

A digression about Functional Programming

A **data type** is an abstraction that encapsulates one reusable coding pattern

Think of **containers** holding your data

Provide clear **semantics** and an **interface** to manipulate the data

Examples

Option

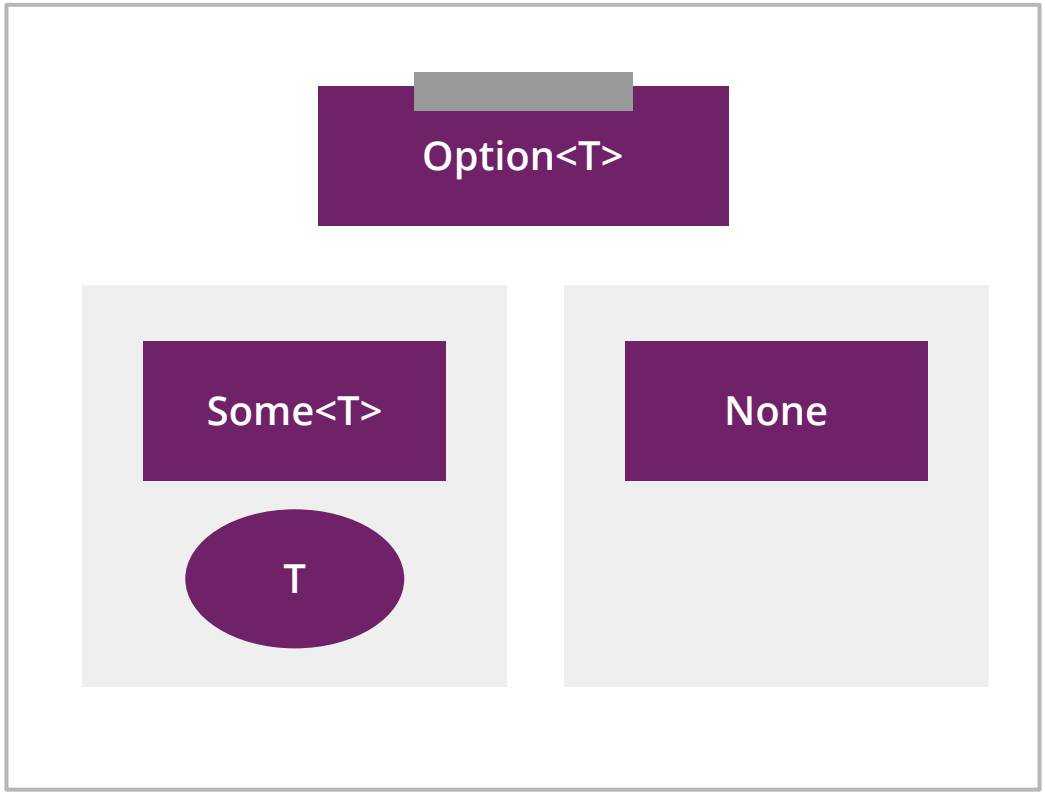
Either

Validated

IO

arrow-kt.io/docs/0.10/datatypes/intro/

A value that might be absent


Option<T>
Some<T>
T
None

```
sealed class Option<out T> {
    data class Some<out T>(val a: T): Option<T>()
    object None: Option<Nothing>()
}
```

```
fun String.extractToken(): Option<String> = startsWith("Bearer ")
        .maybe { split(" ").last() }
```

```
when (val token = header.extractToken()) {
  is Option.None -> ResponseEntity.status(401).build()
  is Option.Some -> ResponseEntity.ok(result.t)
}
```

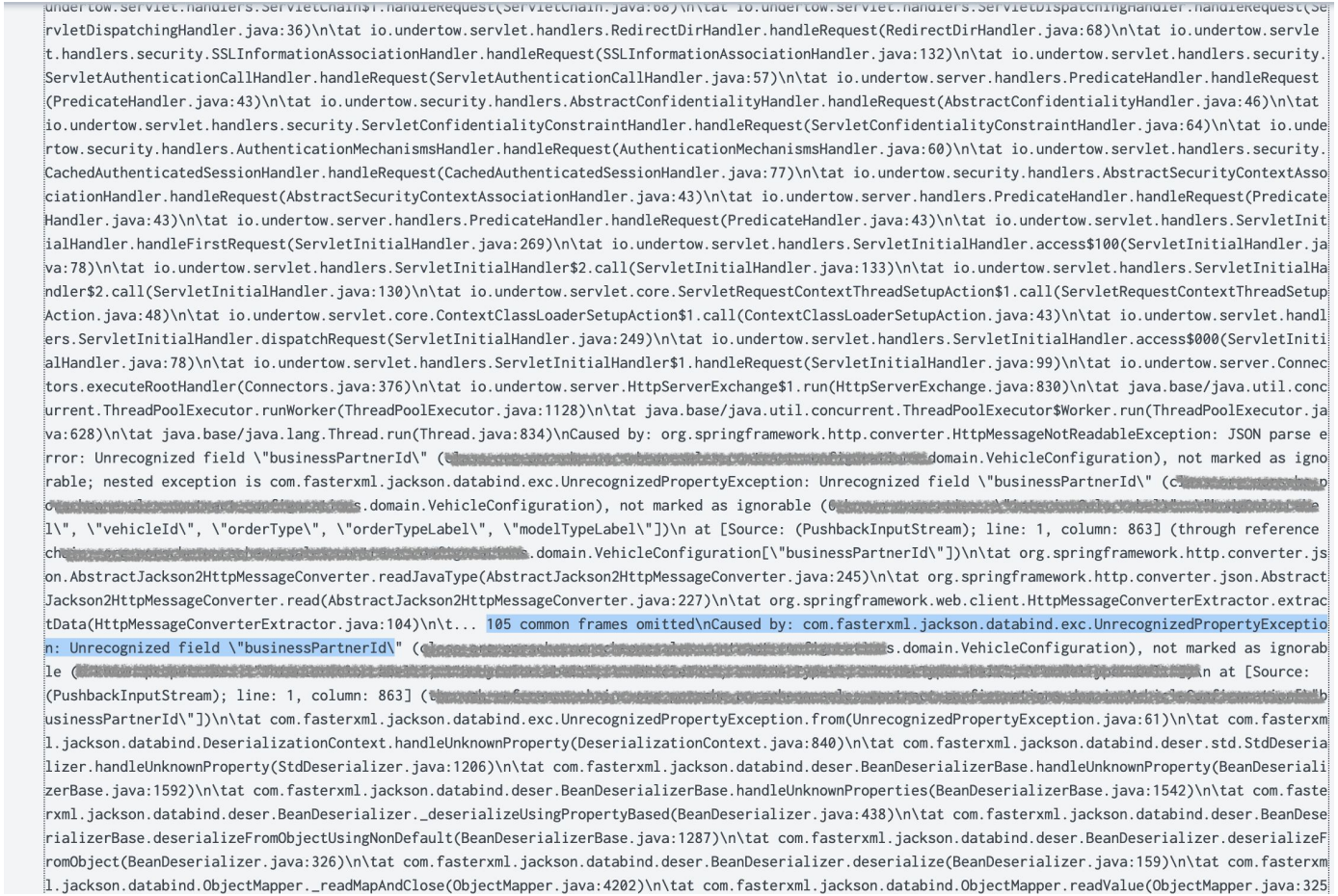
```
interface Operations {
  fun <A, B> Option<A>.map(f: (A) -> B): Option<B>
  fun <A, B> Option<A>.flatMap(f: (A) -> Option<B>): Option<B>
}
```

Exceptions

The hidden GOTO

Did you get a **500** lately in a production system?

… pretty common to throw

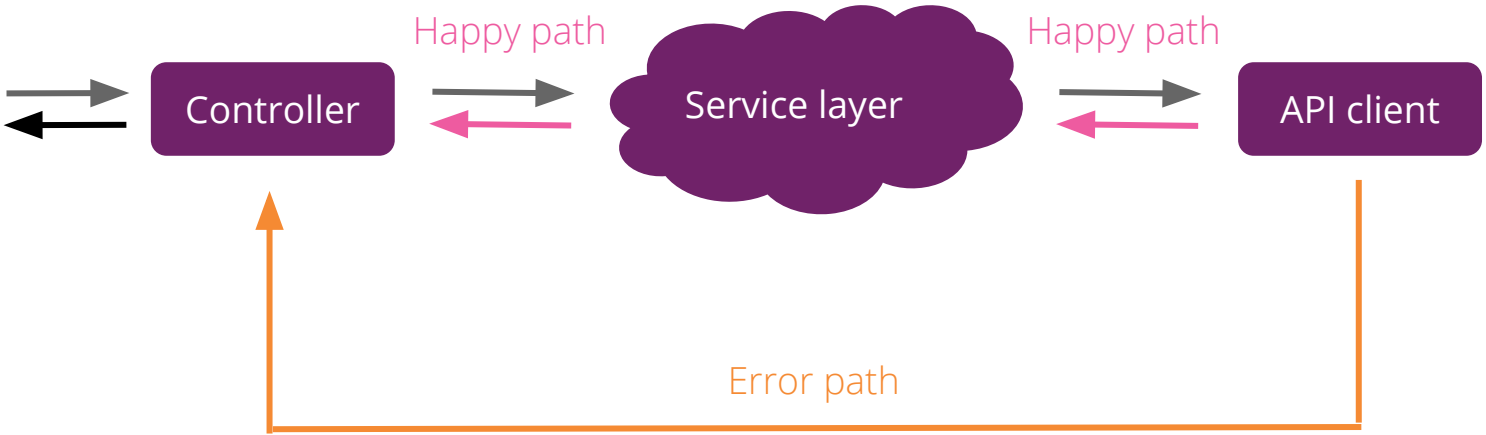


At first Exceptions may sound like a good idea

… but they easily lead to bad decisions

Easy to **ignore** or **miss**

Exceptions oftenly end up being used as **flow control**



```
interface Verifier {
    /**
     * @param jwt a jwt token
     * @return authentication credentials
     */
    fun verify(jwt: String): TokenAuthentication
}
```

```
/**
 * Perform the verification against the given Token
 *
 * @param token to verify.
 * @return a verified and decoded JWT.
 * @throws AlgorithmMismatchException
 * @throws SignatureVerificationException
 * @throws TokenExpiredException
 * @throws InvalidClaimException
 */
public DecodedJWT verifyByCallingExternalApi(String token);
```

```
@ExceptionHandler(JWTVerificationException::class)
fun handleException(exception: JWTVerificationException):
  ResponseEntity<ErrorMessage> {
    return ResponseEntity
      .status(HttpStatus.BAD_GATEWAY)
      .body(ErrorMessage.fromException(exception))
}
```

What is an Exception?

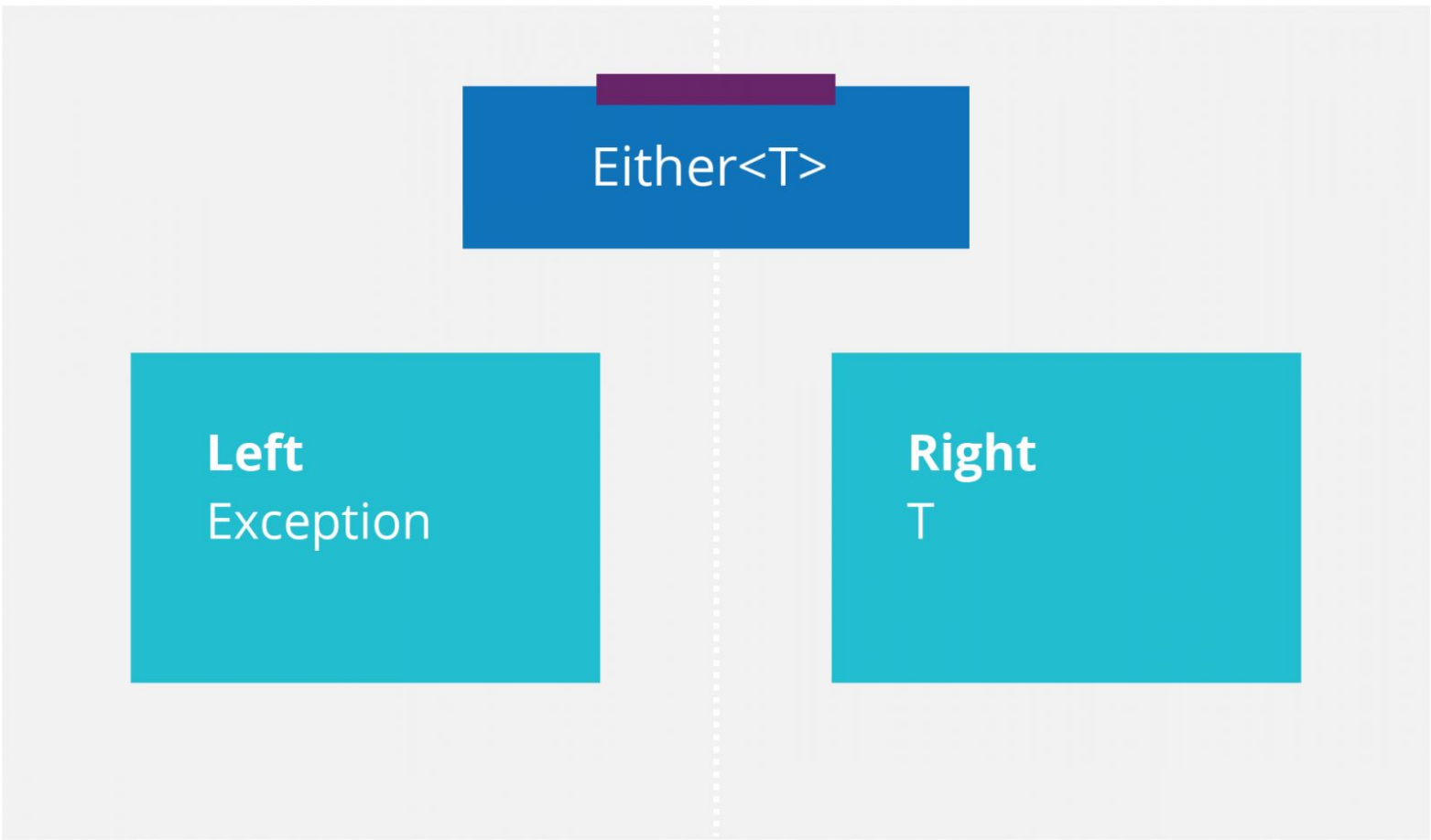
A cluster going down 🔥
is an exception

Not really an exception

As the context grows it becomes harder to **test** and **reason about**

Either

Two different values depending on the result of the computation


Either<T>
Left
Exception
Right
T

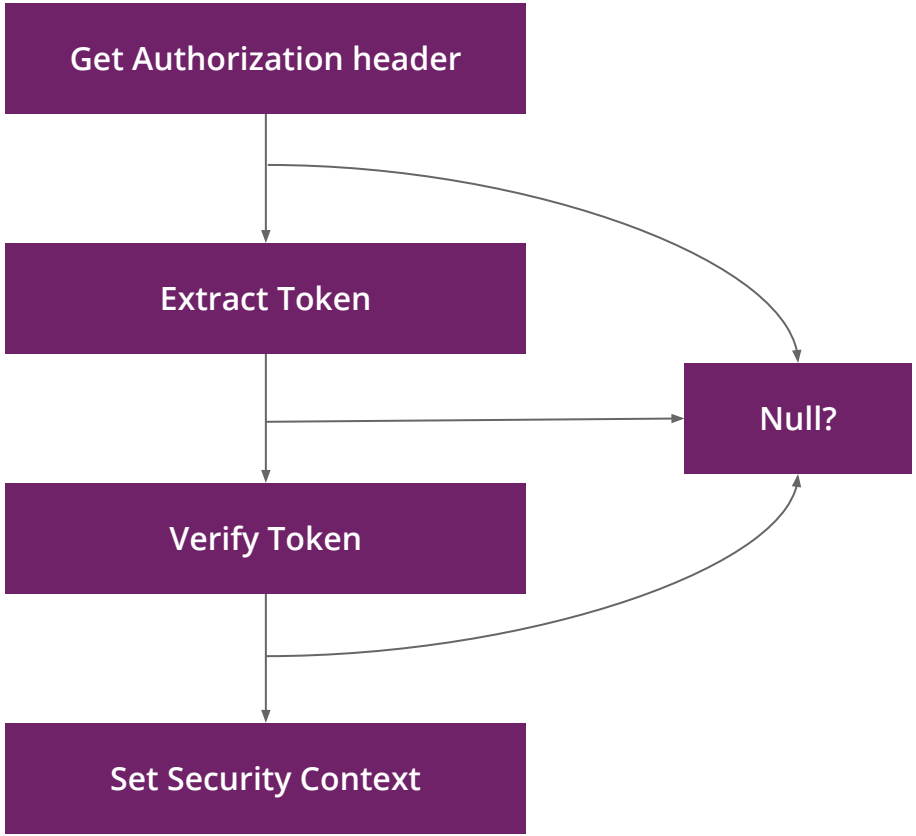
```
sealed class Either<out L, out R> {
  data class Left<out L, out R>(val a: L) : Either<L, R>()
  data class Right<out L, out R>(val b: R) : Either<L, R>()
}
```

```
interface Verifier {
  fun verify(token: String): Either<TokenEx, TokenAuth>
}
```

```
fun Verifier.unsafeVerify(jwt: String): Either<TokenEx, TokenAuth> = try {
  verifyByCallingExternalApi(jwt).right()
} catch (e: JWTVerificationException) {
  e.left()
}
```

```
interface Operations {
  fun <T, A, B> Either<T, A>.map(f: (A) -> B): Either<T, B>
  fun <T, A, B> Either<T, A>.flatMap(f: (A) -> Either<T, B>):
    Either<T, B>
}
```

```
verifier
    .unsafeVerify(jwt)
    .map { it.asToken() }
```


Get Authorization header
Extract Token
Null?
Verify Token
Set Security Context

```
request.getHeader(Headers.AUTHORIZATION)
  .toEither()
  .flatMap { header ->
    header.extractToken()
      .flatMap { jwt ->
        verifier
          .verify(jwt)
          .map { token ->
            SecurityContextHolder.getContext().authentication = token
        }
      }
  }
```

Non-Nested Syntax

Similar to **async/await**

```
Either.fx {
    val (header) = request.getHeader(Headers.AUTHORIZATION).toEither()
    val (jwt) = header.extractToken()
    val (token) = verifier.verify(jwt)
    SecurityContextHolder.getContext().authentication = token
}
```

thoughtworks.com/insights/blog/either-data-type-alternative-throwing-exceptions

Summary

Null and exceptions can lead to flaky, hard to understand code

Let’s stop **using null**

Let’s stop **abusing** **exceptions**

Continue the conversation on Slack

XConfEurope2020
xconfeurope2020.slack.com
#talk4-alternative-to-null-and-exceptions

#XConfOnline