

The Salesforce logo, which consists of a blue cloud-like shape with the word "salesforce" in white lowercase letters inside it.

salesforce

# Handle the Right Errors

Building Resilient and Reliable Apex

David Reed, Lead Member of Technical Staff,  
Release Engineering | [Salesforce.org](https://Salesforce.org)  
[@aoristdual](https://twitter.com/aoristdual) | [d.reed@salesforce.com](mailto:d.reed@salesforce.com)



# David Reed

Lead Member of Technical Staff,  
Release Engineering  
[Salesforce.org](https://salesforce.org)





# Forward-Looking Statement



Statement under the Private Securities Litigation Reform Act of 1995:

This presentation contains forward-looking statements about the company's financial and operating results, which may include expected GAAP and non-GAAP financial and other operating and non-operating results, including revenue, net income, diluted earnings per share, operating cash flow growth, operating margin improvement, expected revenue growth, expected current remaining performance obligation growth, expected tax rates, the one-time accounting non-cash charge that was incurred in connection with the Salesforce.org combination; stock-based compensation expenses, amortization of purchased intangibles, shares outstanding, market growth and sustainability goals. The achievement or success of the matters covered by such forward-looking statements involves risks, uncertainties and assumptions. If any such risks or uncertainties materialize or if any of the assumptions prove incorrect, the company's results could differ materially from the results expressed or implied by the forward-looking statements we make.

The risks and uncertainties referred to above include -- but are not limited to -- risks associated with the effect of general economic and market conditions; the impact of geopolitical events; the impact of foreign currency exchange rate and interest rate fluctuations on our results; our business strategy and our plan to build our business, including our strategy to be the leading provider of enterprise cloud computing applications and platforms; the pace of change and innovation in enterprise cloud computing services; the seasonal nature of our sales cycles; the competitive nature of the market in which we participate; our international expansion strategy; the demands on our personnel and infrastructure resulting from significant growth in our customer base and operations, including as a result of acquisitions; our service performance and security, including the resources and costs required to avoid unanticipated downtime and prevent, detect and remediate potential security breaches; the expenses associated with new data centers and third-party infrastructure providers; additional data center capacity; real estate and office facilities space; our operating results and cash flows; new services and product features, including any efforts to expand our services beyond the CRM market; our strategy of acquiring or making investments in complementary businesses, joint ventures, services, technologies and intellectual property rights; the performance and fair value of our investments in complementary businesses through our strategic investment portfolio; our ability to realize the benefits from strategic partnerships, joint ventures and investments; the impact of future gains or losses from our strategic investment portfolio, including gains or losses from overall market conditions that may affect the publicly traded companies within the company's strategic investment portfolio; our ability to execute our business plans; our ability to successfully integrate acquired businesses and technologies, including delays related to the integration of Tableau due to regulatory review by the United Kingdom Competition and Markets Authority; our ability to continue to grow unearned revenue and remaining performance obligation; our ability to protect our intellectual property rights; our ability to develop our brands; our reliance on third-party hardware, software and platform providers; our dependency on the development and maintenance of the infrastructure of the Internet; the effect of evolving domestic and foreign government regulations, including those related to the provision of services on the Internet, those related to accessing the Internet, and those addressing data privacy, cross-border data transfers and import and export controls; the valuation of our deferred tax assets and the release of related valuation allowances; the potential availability of additional tax assets in the future; the impact of new accounting pronouncements and tax laws; uncertainties affecting our ability to estimate our tax rate; the impact of expensing stock options and other equity awards; the sufficiency of our capital resources; factors related to our outstanding debt, revolving credit facility, term loan and loan associated with 50 Fremont; compliance with our debt covenants and lease obligations; current and potential litigation involving us; and the impact of climate change.

Further information on these and other factors that could affect the company's financial results is included in the reports on Forms 10-K, 10-Q and 8-K and in other filings it makes with the Securities and Exchange Commission from time to time. These documents are available on the SEC Filings section of the Investor Information section of the company's website at [www.salesforce.com/investor](http://www.salesforce.com/investor).

Salesforce.com, inc. assumes no obligation and does not intend to update these forward-looking statements, except as required by law.



# General Principles of Exception Handling

...οὐδὲν γὰρ δεινὸν πείσῃ, ἐὰν τῷ ὄντι ᾗς καλὸς καὶ ἀγαθός, ἀσκῶν ἀρετὴν.

...for you shall suffer nothing terrible,  
if in truth you should be a good and fair person who practices virtue.

– Plato, *Gorgias* 527d

# Why do I need to grapple with this topic?



Every application (and every developer!) starts with the “happy path”: what happens when everything goes to plan.

The path to production is a journey. You’ll learn quickly that some of your plans are wrong, or incomplete.

Exceptions are the friction point between your plans and the real world.

Writing code that degrades gracefully and copes with exceptions and other types of error is a key part of being a developer. It’s part of how you uphold your moral obligations to your users.



New dev tip!





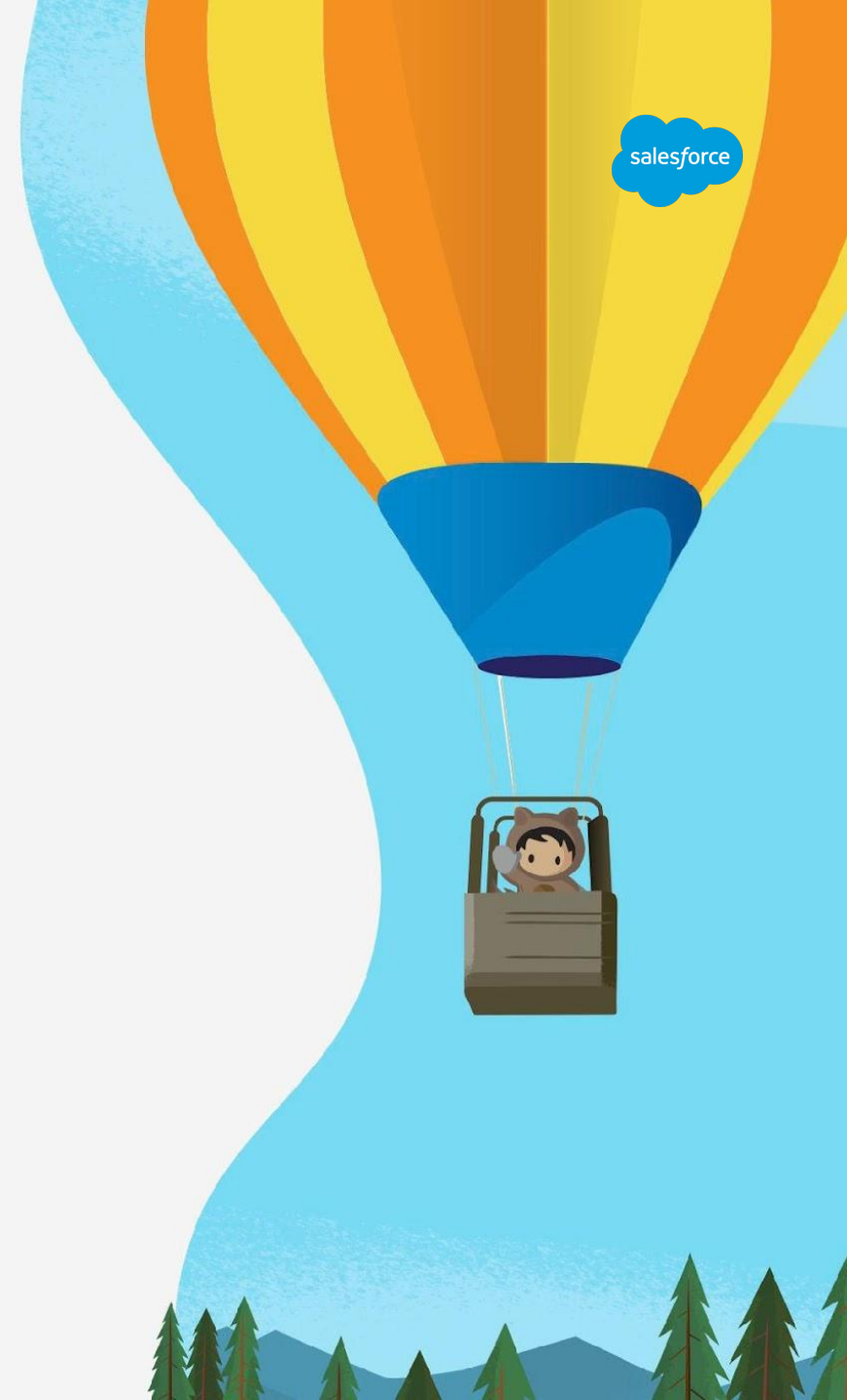
# The notion of handling an exception

Take an *exceptional* situation (something bad and out of the ordinary happened) and allow the application to safely move back into an anticipated pathway of operation.

“Safely” means we preserve

- The integrity of the data involved.
- The experience of the user.
- The outcome of the process, if possible.

usually in that order.



# What counts as exceptional?

The runtime decides, but...

Generally an exception is a response to a situation where correct behavior is **undefined**.

```
Integer a = 0;  
Integer b;  
System.debug(a + b);
```

`b` is `null`, yielding a `NullPointerException`.

There is no sensible path out of this code, so we get an exception.

Resist the urge to see exceptions as impediments. They're aids.



New dev tip!



# What if I do nothing?

## Rollback!

If you do nothing, and an exception is raised, all DML and DML-equivalent actions are rolled back for the entire transaction.

- Inserting, updating, deleting records
- Enqueueing Asynchronous Apex (batch, Queueable, future)
- Sending outbound email

You can't persist any data or functionality from a rolled-back transaction.\*

Rollbacks preserve the integrity of the database.



New dev tip!





# What counts as handling?

At least five different strategies

1. Correcting the conditions that produced the exception and continuing down the original code path.
2. Switching to an alternate code path.
3. Catching the exception at the transaction entry point to provide clear messaging.
4. Swallowing the exception and logging it for asynchronous resolution.
5. Terminating the transaction and rolling back all changes.

Real code is complex and doesn't always fall into neat ontologies. Use these patterns as tools to discover the best solution for a given case, not hard and fast rules.



New dev tip!



What is at stake here?



USER  
TRUST

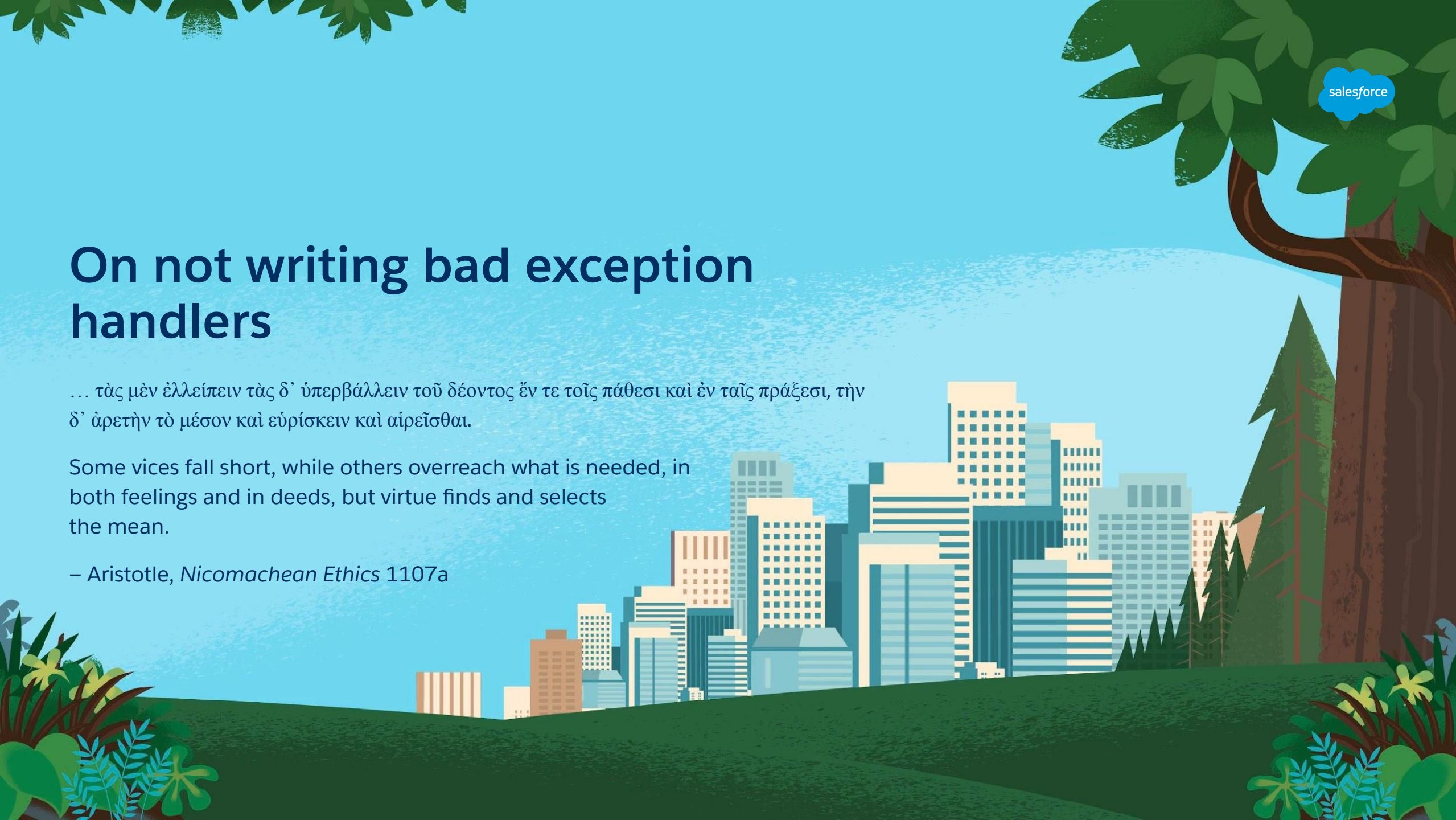


# On not writing bad exception handlers

... τὰς μὲν ἐλλείπειν τὰς δ' ὑπερβάλλειν τοῦ δέοντος ἔν τε τοῖς πάθεσι καὶ ἐν ταῖς πράξεσι, τὴν δ' ἀρετὴν τὸ μέσον καὶ εὕρισκεν καὶ αἰρεῖσθαι.

Some vices fall short, while others overreach what is needed, in both feelings and in deeds, but virtue finds and selects the mean.

– Aristotle, *Nicomachean Ethics* 1107a





# Exception handling over logic

Fine in Python, not fine in Apex (Strategy 1)

```
try {
    processData(someMap.get(c.AccountId).Parent);
} catch (NullPointerException e) {
    // do nothing
}
```

## Why is it no good?

It's not idiomatic Apex. It's not performant. It's harder to test.



# Exception handling over logic

Fine in Python, not fine in Apex

```
if (someMap.containsKey(c.AccountId)) {  
    processData(someMap.get(c.AccountId).Parent);  
}
```

## Why is this better?

It's idiomatic. It's performant. It's easy to test.

“Look before you leap.”



# Unneeded exception handlers

Understandable, but wrong (Strategy 1/2)

```
List<Account> accountList;  
try {  
    accountList = [SELECT Id, Name FROM Account];  
} catch (Exception e) {  
    accountList = new List<Account>();  
}
```

## Why is it no good?

- There's no catchable exception here!
- Increases complexity of code
- Deceives the reader
- Impossible to test.





# Unneeded exception handlers

Understandable, but wrong

```
accountList = [SELECT Id, Name FROM Account];
```

Why is it better?

It's clearer, more understandable, and can be tested.

**Remember:** Limits-related exceptions are not catchable.



New dev tip!



# Overbroad exception handlers

Why is this code wrong? (Strategy 3)

```
try {  
    List<Account> accountList = [  
        SELECT Id, Name, Parent.Name  
        FROM Account  
        WHERE Name = :someAccountName  
        FOR UPDATE  
    ];  
    if (accountList[0].Parent.Name.contains('ACME')) {  
        accountList[0].Description = 'ACME Subsidiary';  
    }  
    update accountList;  
} catch (Exception e) {  
    throw new AuraHandledException(  
        'Could not acquire a lock on this account'  
    );  
}
```



# Overbroad exception handlers

## Masking other errors

This code can throw three catchable exceptions *other* than the `QueryException` we're looking for: `NullPointerException`, `ListException`, and `DmlException`.

```
} catch (QueryException e) {  
    throw new AuraHandledException(  
        'Could not acquire a lock on this account'  
    );  
}
```

## Why is it better?

Catch *specific* exceptions you know could be thrown to avoid hiding unrelated issues in your code.

Design unit tests to be thorough and provoke unexpected exceptions.





# Failing to handle side effects

What's in the database if the second DML throws an exception? (Strategy 3/4)

```
public static void commitRecords(  
    List<Account> accounts, List<Opportunity> opportunities  
) {  
    try {  
        insert accounts;  
        insert opportunities;  
    } catch (DmlException e) {  
        ApexPages.addMessage(new  
ApexPages.Message(ApexPages.Severity.FATAL, 'Unable to  
save the records');  
    }  
}
```



# Failing to handle side effects

What's the state of the database if the second DML throws an exception?

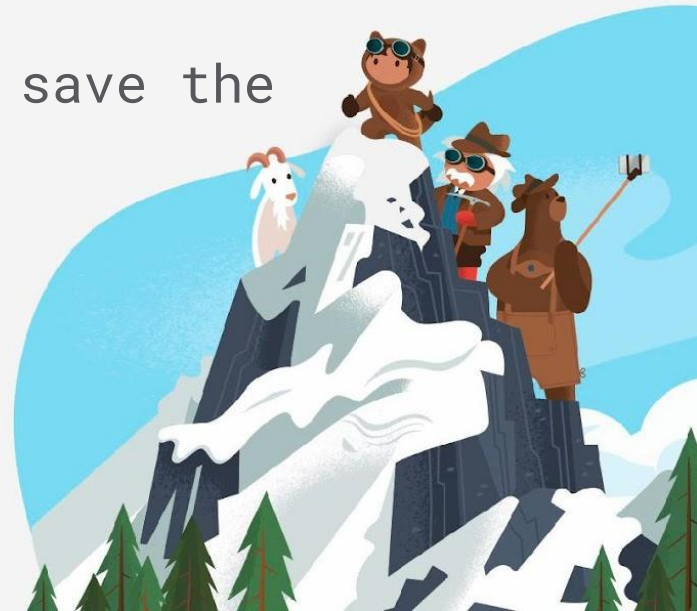
- The Accounts remain inserted.
- The Opportunities do not.
- The state of the Visualforce controller (or other controlling code) may be inconsistent.
- Retrying the action or standing pat *both* may leave the database in (different) bad states.



# Failing to handle side effects

Use explicit rollbacks to preserve data integrity if not re-raising (Strategy 3/4)

```
public static void commitRecords(  
    List<Account> accounts, List<Opportunity> opportunities) {  
    Database.Savepoint sp = Database.setSavepoint();  
    try {  
        insert accounts;  
        insert opportunities;  
    } catch (DmlException e) {  
        ApexPages.addMessage(new  
ApexPages.Message(ApexPages.Severity.FATAL, 'Unable to save the  
records');  
        Database.rollback(sp);  
    }  
}
```





# Failing to handle side effects

Use explicit rollbacks to preserve data integrity if not re-raising (Strategy 3/4)

Why is it better?

- Database returns to its initial state.
- User can retry their initial action after correcting problems
- Integrity is preserved.




# The single worst thing you can do

Strategy 4 - but a useless, damaging implementation

```
try {
    // Do some complex code here
} catch (Exception e) {
    System.debug(e.getMessage());
}
```

1. The failure has no user-facing consequences.
2. The error does not harm data integrity or operations.
3. The error doesn't need to be corrected.

If those three things are true,  
**why are you writing this code?**



What has to be true for  
this pattern to be  
correct?

# Where's Strategy 5?



Just do nothing.

The platform handles rollbacks for you, or a higher tier of code takes Strategy 3.

Yes, the user might get an ugly error message - but the integrity of the database is protected.

If you're in lower-level code, that's your concern. Let the controller or entry point layer deal with presentation.



# On writing good exception handlers

ἐὰν μὴ ἔλπηται ἀνέλπιστον, οὐκ ἐξευρήσει, ἀνεξερεύνητον ἐὼν καὶ ἄπορον

Should one not expect the unexpected, one shan't find it, as it is hard-sought and trackless.

– Heraclitus, Fragment 18



# Questions to start with



- What is the state of the database before and after the error?
- What layer of functionality are we operating in?
- What would it mean to preserve our three objectives in this context?
  - The integrity of the data involved.
  - The experience of the user.
  - The outcome of the process, if possible.

USER  
TRUST



# What counts as handling?

At least five different strategies

1. Correcting the conditions that produced the exception and continuing down the original code path.
2. Switching to an alternate code path.
3. Catching the exception at the transaction entry point to provide clear messaging.
4. Swallowing the exception and logging it for asynchronous resolution.
5. Terminating the transaction and rolling back all changes.

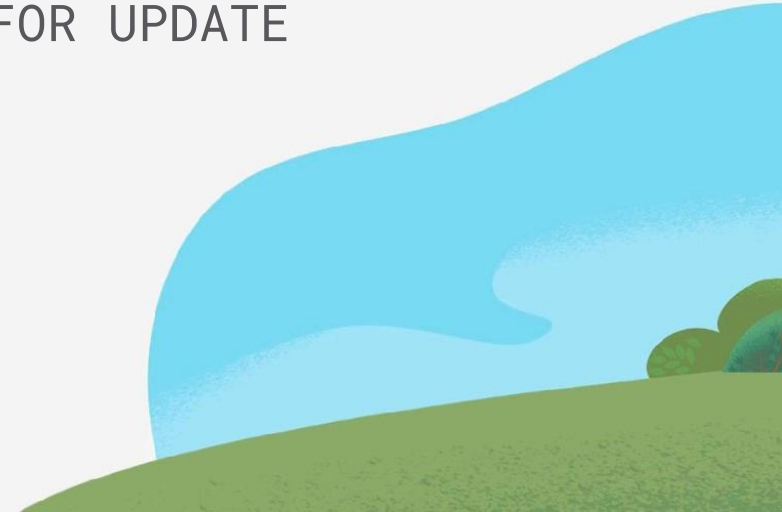


# Correcting the exceptional condition



- Key question: can this error be fixed in Apex without user input?
- Relatively uncommon situation. You'd generally fix the issue *before* an exception occurs.
- Example: Queueables recursing to wait out a transaction lock.

```
public void execute(QueueableContext qc) {  
    try {  
        List<Account> a = [  
            SELECT Id FROM Account WHERE Id = :someId FOR UPDATE  
        ];  
    } catch (QueryException e) {  
        System.enqueueJob(new MyQueueable(someId));  
    }  
}
```



# Switching to alternate code path

- Key question: can I supply a sensible default or alternate route?

```
Fund__c f;  
try {  
    f = [SELECT Id FROM Fund__c WHERE Default__c = true LIMIT 1];  
} catch (QueryException e) {  
    f = new Fund__c(Name = 'Default', Default__c = true);  
    insert f;  
}
```

*Note: this example violates the principle of “look before you leap!”*



# Catching at the entry point

- Key question: what code layer am I in?
- Wrapping the exception at the entry point to provide clear messaging.
- Potential loci:
  - Lightning Apex controller: catch and rethrow `AuraHandledException`
  - Visualforce controller: catch, rollback, show page message
  - Web service or REST API endpoint: catch, rollback, return clear error.



# Catching at the entry point

## Visualforce example

```
Database.Savepoint sp = Database.setSavepoint();
try {
    doSomeVeryComplexOperation(myInputData);
} catch (DmlException e) {
    Database.rollback(sp);
    ApexPages.addMessage(
        new ApexPages.Message(ApexPages.Severity.FATAL, 'Unable to save
the data. The following error occurred: ' + e.getMessage()));
} catch (CalloutException e) {
    Database.rollback(sp);
    ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.FATAL,
'We could not reach the remote system. Please try again in an hour.'));
}
```



# Swallowing and logging for async resolution

## Strategy 4

- Key questions
  - Does my handling decision impact data integrity?
  - Can this error be acted upon by an admin if logged and surfaced?
  - Do I have effective reporting to surface this log?
- `System.debug( )` does not count.
- Examples: batch class updating data; outbound email notifications.



# Swallowing and logging for async resolution

Example: batch class updating data via callout

```
public void execute(Database.BatchableContext bc, List<Account> scope) {
    for (Account a : scope) {
        try {
            updateAccountDataViaCallout(a);
        } catch (CalloutException e) {
            insert new Error_Log__c(
                Context__c = 'BATCH', Record_Id__c = a.Id, ...
            );
        }
    }
    update scope;
}
```

*This is a simplified example. Use an error logging framework.*





# “But I need to log *all* errors!”

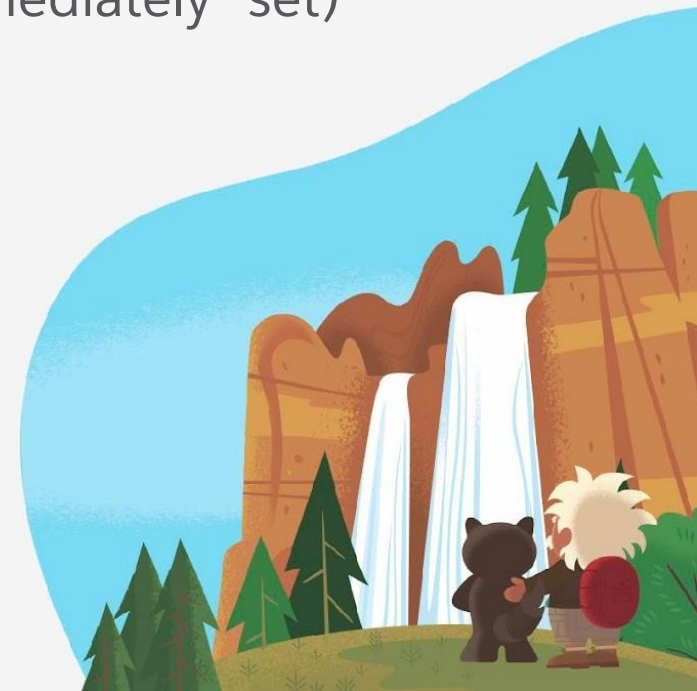
Use a pattern that genuinely achieves that end (Strategy 4)

Swallowing errors for the purpose of logging them is often the wrong strategy if what you really need is a rollback.

If you want to log errors, do so in a way that meets the three objectives.

Options include:

- Non-transactional functionality (Platform Events with “Publish Immediately” set)
- Manual rollbacks + DML on error log
  - Is the error actionable?
  - Am I at the entry point, where I can effectively manage rollback of all involved data?
- Swallow and log where
  - Integrity is already broken and can’t be restored.
  - Functionality is non-critical but needs to be repaired.



# Terminating and rolling back

This is the easy one

Just do nothing! Let the higher tier functionality make the decision.

*or*, in a trigger context, apply partial success DML methods and use `addError()` appropriately to delegate responsibility up to a higher tier.



# Last thoughts: testing exception handlers



- Testing exception handlers often requires dependency injection.
- The exercise of writing handler tests can illuminate incorrect assumptions.
  - Is it even theoretically possible to produce the entry conditions for this handler?



New dev tip!

```
try {  
    accountList = [SELECT Id, Name FROM Account];  
} catch (Exception e) {  
    accountList = new List<Account>();  
}
```

# Trivia!



What is the complete list of exceptions this code can throw?

```
public void updateAccountWithPrimaryContact(String actName, String ctId) {  
    Id contactId = (Id)ctId;  
    List<Account> accountList = [  
        SELECT Id, Name, Parent.Name FROM Account WHERE Name = :someAccountName  
        FOR UPDATE  
    ];  
    if (accountList[0].Parent.Name.contains('ACME')) {  
        accountList[0].Primary_Contact__c = contactId;  
    }  
    update accountList;  
}
```





thank  
you

BLAZE  
YOUR  
TRAIL

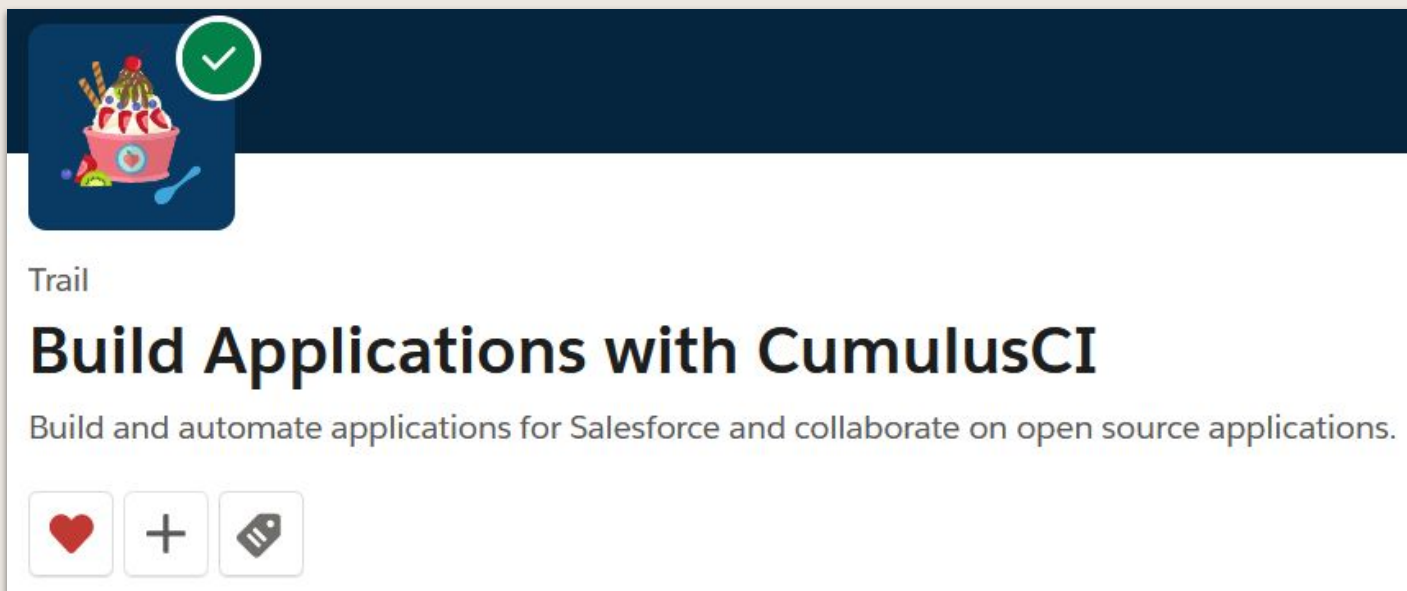
salesforce

# Plug: Complete our Trail!



## Build Applications with CumulusCI:

<https://trailhead.salesforce.com/en/content/learn/trails/build-applications-with-cumulusci>



The image shows a digital screen displaying a Trail card. The card has a dark blue header with a green checkmark icon in a white circle. Below the header is a white section with the word "Trail" in small text, followed by the title "Build Applications with CumulusCI" in large bold text. Under the title is a subtitle: "Build and automate applications for Salesforce and collaborate on open source applications." At the bottom of the card are three icons: a red heart, a plus sign, and a tag icon.

Trail

### Build Applications with CumulusCI

Build and automate applications for Salesforce and collaborate on open source applications.

♥ + 🏷️