



Exception Handling Fundamentals & Advanced Techniques



Introduction & Importance



```
42 // Sign user up
43 await Provider.of<Auth>(context, listen: false).signUp(
44   _authData['email'],
45   _authData['password'],
46 );
47 }
48 } on HttpException catch (error) {
49   var errorMessage = "Recheck Credentials";
50   if (error.toString().contains("EMAIL_EXISTS")) {
51     errorMessage = "Email already in use";
52   } else if (error.toString().contains("INVALID_EMAIL")) {
53     errorMessage = "Please Entry Correct Email Address Formation";
54   } else if (error.toString().contains("WEAK_PASSWORD")) {
55     errorMessage = "Password should be at least 8 Characters";
56   } else if (error.toString().contains("INVALID_PASSWORD")) {
57     errorMessage = "Password is not correct";
58   } else if (error.toString().contains("EMAIL_NOT_FOUND")) {
59     errorMessage = "Give the correct Email please";
60   } else {
61     errorMessage = "FUCK";
62   }
63   print(errorMessage);
64   _showErrorDialog(errorMessage);
65 }
66 catch (error) {
67   const errorMessage = "Couldn't Authenticate You !";
```

What are exceptions?

- Anomalous conditions that disrupt program flow.
- Can be caused by various factors (e.g., user input, resource errors, network issues).

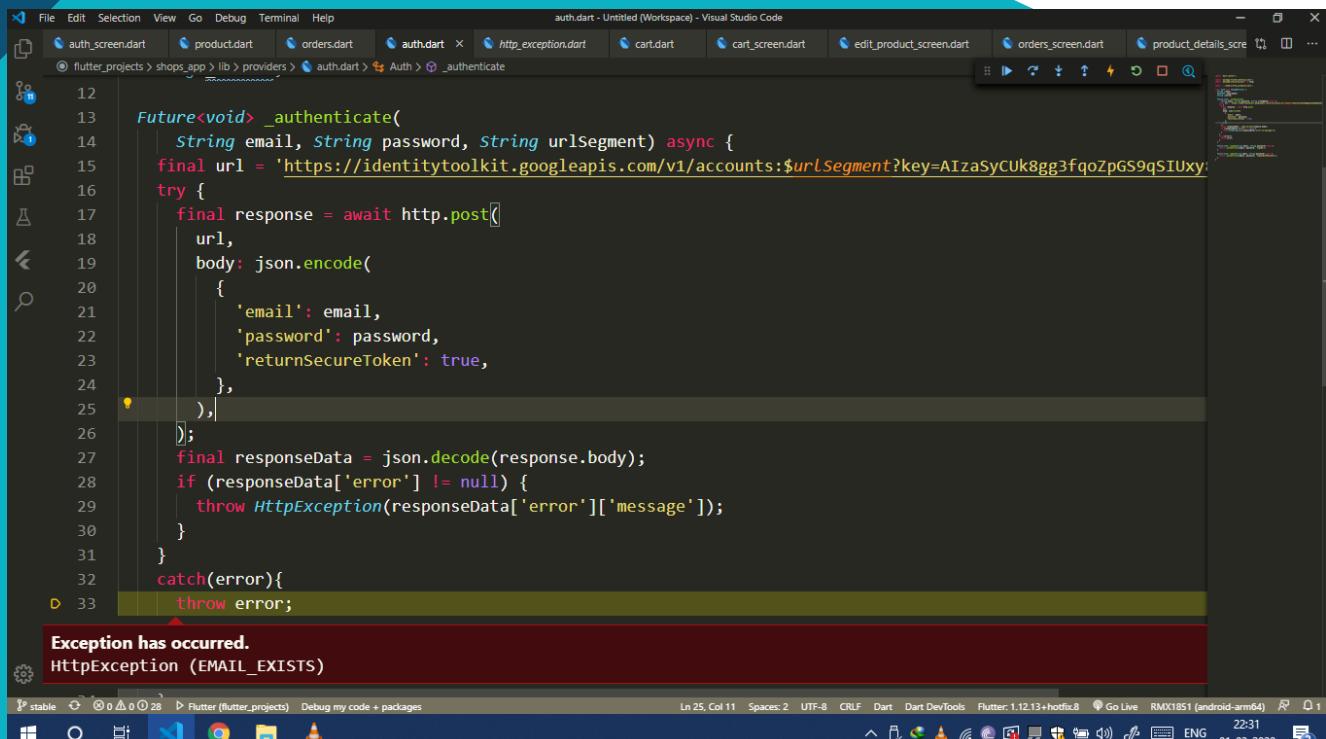
Importance of exception handling

- Ensures program stability and prevents crashes.
- Allows for graceful handling of errors.
- Improves application robustness and user experience.

Benefits of proper exception handling

- Increased code quality and maintainability.
- Easier debugging and troubleshooting.
- Improved error reporting and logging.
- Provides a more predictable and reliable program execution.

Fundamentals - Terminology & Types



Key terms:

- try-catch block
 - finally block
 - throw
 - throws

Types of exceptions

- Checked
 - Unchecked

Handling Exceptions

Specific vs. General exceptions:

- Specific: targeted handling for defined exceptions (e.g., `FileNotFoundException`).
- General: catch-all for unexpected or unknown errors.

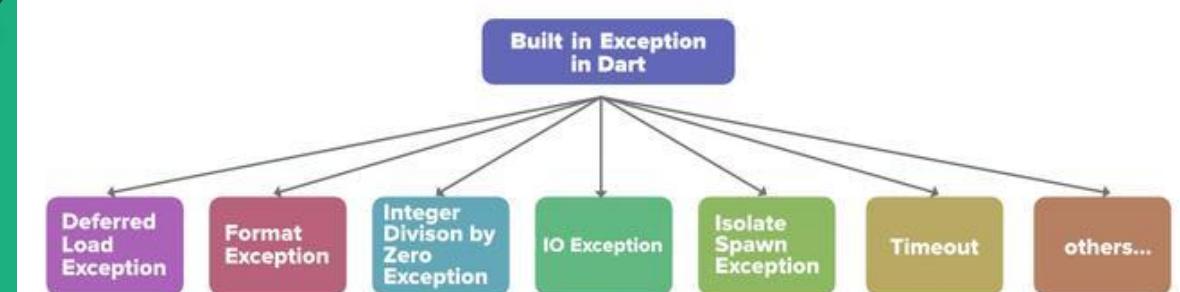


Creating custom exceptions:

- Define your own exceptions for specific situations.
- Provides more granular error handling and information.

Rethrowing exceptions:

- Propagate an exception caught in a catch block.
- Allows handling the exception at a higher level.



Advanced Techniques - Nested Blocks & Chained Exceptions

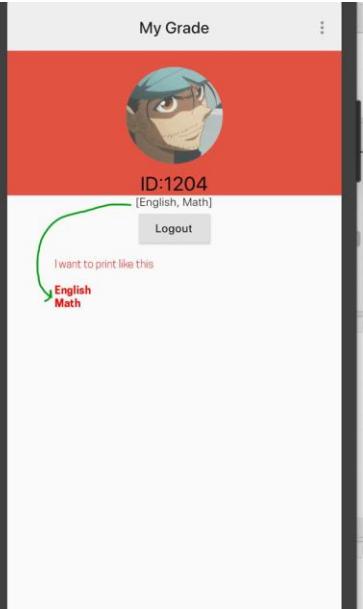
Nested try-catch blocks:

- Multiple try-catch blocks nested inside each other.
 - Provides finer-grained control over exception handling for specific code sections.



Chained exceptions:

- Link exceptions together to trace their origin and cause.
 - Helpful for debugging and understanding the root cause of an error.



Advanced Techniques - Multithreading



```
import 'dart:async';

Future<void> computeFunction() async {
  int result = await compute(computeIntensiveFunction, 100);
  print(result);
}

int computeIntensiveFunction(int value) {
  // Perform some compute-intensive task
  return value * value;
}
```

Exception handling in multithreaded environments:

- Synchronize access to shared resources to avoid race conditions.
- Ensure thread safety when handling exceptions.

Real-world Examples & Pitfalls

- Demonstrate different exception handling techniques in practical situations.
- Discuss potential pitfalls and solutions for common errors.



No Material widget found.
TextField widgets require a Material widget ancestor.
In Material Design, most widgets are conceptually "printed" on a sheet of material. In Flutter's material library, that material is represented by the Material widget. It is the Material widget that renders ink splashes, for instance.
Because of this, many material library widgets require that there be a Material widget in the tree above them.
To introduce a Material widget, you can either directly include one, or use a widget that contains Material itself, such as a Card, Dialog, Drawer, or Scaffold.
The specific widget that could not find a Material ancestor was:
TextField

Best Practices & Guidelines



Principles for robust exception handling:

- Follow consistent and clear coding patterns.
- Use descriptive exception messages.
- Design for testability and maintainability.

Choosing the right approach:

- Consider the context and severity of potential exceptions.
- Choose appropriate try-catch blocks and exception types.
- Balance clarity with efficiency.