

Design Patterns

Intro

<https://codewithmosh.com/courses/enrolled/759570> a good course

Design patterns are elegant solutions to common repeating problems in software design. A good book proposed by many as a reference: Design patterns elements of reusable object oriented software, by GoF. The book separates 23 design patterns in three categories, **creational** (patterns that deal with different ways to create objects), **structural** (patterns that define the relationships between these objects) and **behavioral** (deal with the communication between these objects). Design Patterns help you build reusable, extensible and maintainable software

Patterns to have in mind

Memento, state, command, observer, mediator, visitor, pipeline,

The 4 basic principles of OOP

encapsulation, abstraction, inheritance and polymorphism.

<pre>package com.codewithmosh; public class Account { private float balance; public void setBalance(float balance) { if (balance > 0) this.balance = balance; } public float getBalance() { return balance; } }</pre>	Encapsulation Balance can't be accessed from outside of this class. The outsiders only use setBalance and getBalance. (python's property method is a nice syntactic sugar for this)
---	--

<pre>public class MailService { public void sendEmail() { connect(timeout: 1); authenticate(); // Send email disconnect(); } private void connect(int timeout) { System.out.println("Connect"); } private void disconnect() { System.out.println("Disconnect"); } private void authenticate() { </pre>	Abstraction Reduce complexity by hiding unnecessary details in the classes. We hide connect, authenticate and disconnect from the rest of the program by making them private. We only need to interact with the send_email method of this class. The rest of the methods are implementation details of this class.
---	--

	Inheritance
--	--------------------

```

package com.codewithmosh;

public class Main {
    public static void main(String[] args) {
        drawUIControl(new CheckBox());
    }

    public static void drawUIControl(UIControl control) {
        control.draw();
    }
}

package com.codewithmosh;

public abstract class UIControl {
    public void enable() {
        System.out.println("Enabled");
    }

    public abstract void draw();
}

package com.codewithmosh;

public class CheckBox extends UIControl {
    @Override
    public void draw() {
        System.out.println("Drawing a checkbox");
    }
}

```

Polymorphism

An object (an interface) can take many different forms (implementations of the interface).

It uses the concept of interfaces. You work with an interface and not the actual implementation of the interface.

In this case you have an abstract method UIControl (the interface) which is an abstract method for ui buttons of a menu. Then you have an actual implementation which is the CheckBox. In the main program you use the UIControl (the functions work with an UIControl instance) and not a specific implementation. So you just call the draw method of the interface which is implemented differently in each implementation.

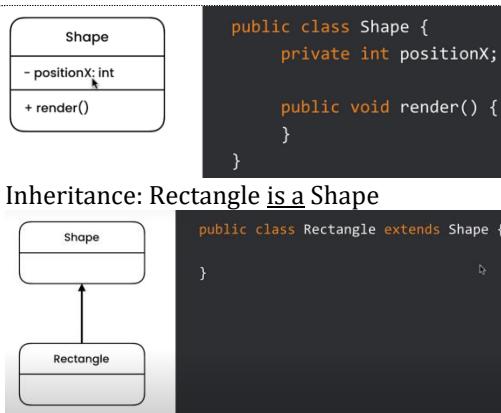
OOP principles

- Single Responsibility Principle
- Open Closed Principle



By following this principle, you can add new functionality without changing existing code. For example when we use a generic interface with specific implementations of it, in order to add new functionality, we just have to add a new interface implementation without modifying existing code. This makes our codebase more robust to changes, easily extensible and maintainable.

UML

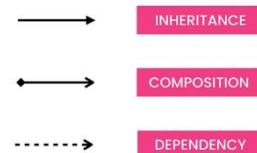


Composition: Shape has a Size

Unified modelling language. A graphical notation to represent a system (its classes and their dependencies).

Without parenthesis we define the variables and with () the methods of a class.

We have 3 types of relationships between classes: inheritance, composition and dependency relationship

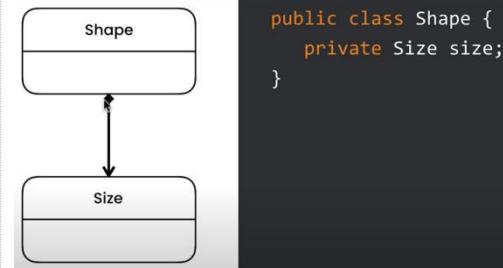


Dependency

Somewhere in the shape class we have a reference (a dependency) to the document class

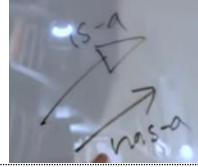
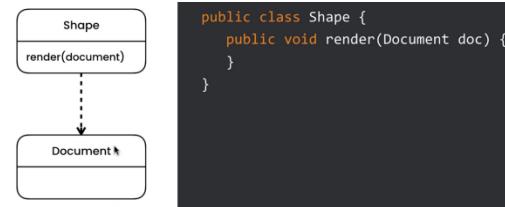
The inheritance arrow (it is empty) means "is a" and the composition "has a".

Dependency occurs when one object "is dependent" on another. It can occur with or without a relation between the 2 objects. Actually, one object might not even be knowing that another exists, yet they might be dependent. Example : The



Dependency: Shape depends on (references) Size

Document



```

public class Shape {
    private Size size;
}
  
```

Producer-Consumer problem. The producer need not know that the consumer exists, yet it has to do wait() and notify(). So, "NO", dependency is not a subset of association.

Composition : (when an object's variable is another object, self.other_object = other_object). Is a type of association in which the "child" object cannot exist without the parent class. i.e, if the child object exists, then it MUST BE IN THE parent Object and nowhere else. EG: A Car (Parent) has Fuel injection system (child). Now, it makes no sense to have a Fuel Injection system outside a car (it will be of no use). i.e, Fuel injection system cannot exist without the car.

Aggregation : Here, the child object can exist outside the parent object. A Car has a Driver. The Driver CAN Exist outside the car.

Patterns

--- Behavioral ---

They deal with the communication between objects

Memento

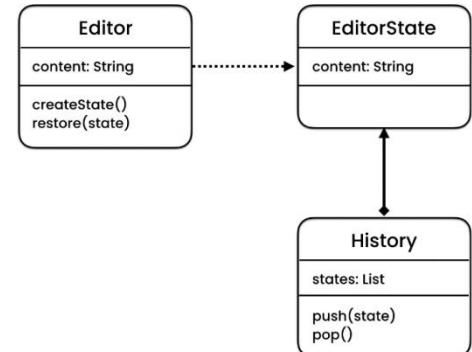
```

Main.java
Editor.java
package com.codewithmosh.memento;

public class Editor {
    private String content;

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
  
```



It's a pattern for implementing undo functionality.

In this case we have an editor that shows some content and we want to implement an undo functionality. The editor is a class with one variable, the content (with setter and getter methods). how would you implement an undo behaviour? One solution would be to use a list that stores the previous contents. Notice though, that later we could add more variables to the editor that need to be "undone" apart from content, for example title. This would mean that we now have to use many lists for storing many variables histories which is not convenient. So we can have all the undoable variables as keys in a state object, and store a list of state's history.

In the memento pattern, we create a class that represents the state of the editor (EditorState). its members are the undoable variables. So each editor state (snapshot), is an instance of this class. Now we need to have a list of those instances. We use another class to store that list (the History class). Each instance of this class is a history of states (snapshots). (We say that the history class is composed of EditorState). The Editor class has a createstate method that creates a new state instance. Then it pushes that state instance to a history instance.

Notice the difference with the undo capability of the command pattern. In the memento pattern even if only one variable of the state changes, we still create a new state instance which contains all the state variables. So we keep a history of entire state objects even if only one variable of the state changes. This has memory implications. In the command object we would have commands for each variable change. For example, changeContentCommand, changeTitleCommand etc. Then each individual action could be undoable. Each action could be stored in global history object. When you undo, you pop a command from history and execute its unexecute method. Notice that in memento the history object holds a list of states while in the command pattern a list of commands (that know how to unexecute themselves).

Me: we can make the history object an Iterator (see the iterator pattern).

```

public class Main {
    public static void main(String[] args) {
        var editor = new Editor();
        var history = new History();

        editor.setContent("a");
        history.push(editor.createState());

        editor.setContent("b");
        history.push(editor.createState());

        editor.setContent("c");
        editor.restore(history.pop());

        System.out.println(editor.getContent());
    }
}

```

State

It is called state pattern, because it allows an object to behave differently if the state changes. For example, the mouseUp method behaves differently depending on the state of the Tool interface.

```

public class Main {
    public static void main(String[] args) {
        var editor = new Editor();
        var history = new History();

        editor.setContent("a");
        history.push(editor.createState());

        editor.setContent("b");
        history.push(editor.createState());

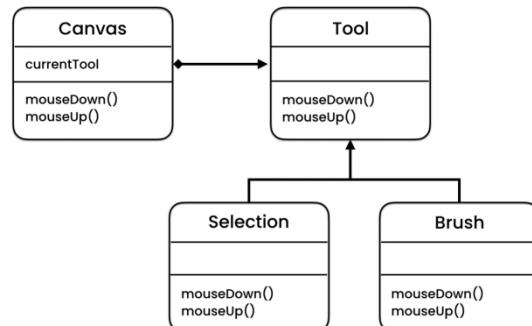
        editor.setContent("c");
        editor.restore(history.pop());

        System.out.println(editor.getContent());
    }
}

```

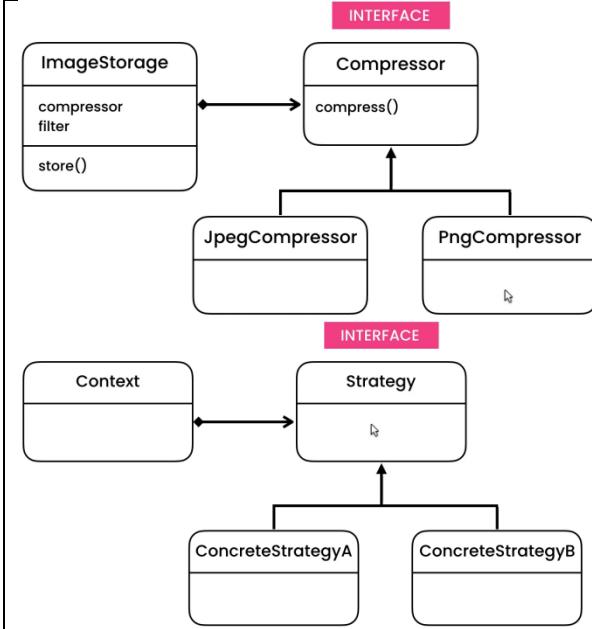
Say you want to build an app with various tools available to the user (brush, eraser, drawer etc.). When you select a tool the mouse down and drag functionality changes (the drawer draws a line, the eraser erases etc.). one way to implement this is to check with if else statements on each mouse action or keyboard button. This means that you will repeat this if else statements many times within your app (on mouseUp, mouseDown, on key pressed and other similar functions) making it eventually hard to change and maintain.

Instead we use polymorphism. We implement a button interface as an abstract class and each button is a different implementation of this interface. In the code we work with the interface. Whenever we want to add a new tool we just have to implement a new implementation of the interface. This approach follows the open closed principle.



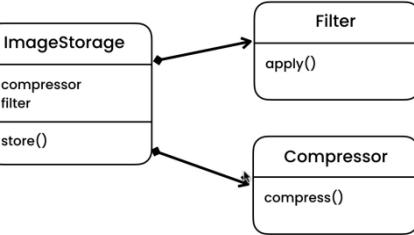
Strategy

It's actually the state pattern for more than one variables. This pattern promotes composition over inheritance.



We want to store an image. Before storing, the image must be compressed and filtered. We can have more than one compressing and filtering algorithms. Instead of creating if else loops we use interfaces.

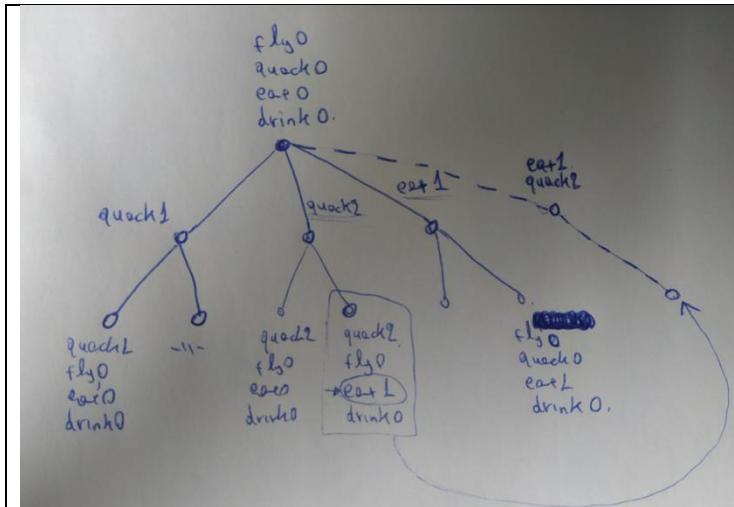
It looks a lot like the state pattern but there is a difference. In the state pattern, the object (canvas) can have a single state (the current tool) that defines all its methods (mouseDown and mouseUp). On the contrary in the Strategy pattern we don't have a single state. We have multiple "states" or behaviors (compressor, filter) represented by one Interface each.



(notes from another source)

It is a pattern that describes a composition mechanism. The **imageStorage** class is composed of (or has) a **compressor** (an abstract compressor behavior) and a **filter** (an abstract filter behavior). Then a specific compressor behavior extends the abstract compressor behavior. In the generic notation, the **Context** is composed of **Strategies**.

This pattern promotes composition over inheritance. Think of the example with the ducks. Instead of using inheritance which leads to the problem where same level subclasses can't easily share logic, we use composition, where we implement some abstract interfaces (the strategy interfaces). Each abstract strategy interface can be implemented by a number of strategies that contain the actual implementation. This way instead of having separate subclasses (for example a mountain duck that inherits from duck) the subclass is replaced by a specific configuration of strategies.



If we have many variants of ducks, we could not have a clean inheritance hierarchy. Instead, we might need some combinations of properties that belong to different subclasses "of the same level". In these cases it's better to use composition using the strategy design pattern where the duck class uses strategy interfaces (duck behaviors). A duck behavior can have various implementations (actual strategies) where each strategy has distinct method implementations. So now a duck is a configuration of strategies instead of a specific subclass.

In this picture we would need to create a new subclass that implements the **eat1** and **quack2** methods. If we have more variations we would need more subclasses.

```

from abc import ABC, abstractclassmethod

class Duck:
    def __init__(self, fly_behaviour, quack_behaviour, display_behaviour):
        self.fly_behaviour = fly_behaviour
        self.quack_behaviour = quack_behaviour
        self.display_behaviour = display_behaviour

    def fly(self):
        return self.fly_behaviour.fly()

    def quack(self):
        return self.quack_behaviour.quack()

    def display(self):
        return self.display_behaviour.display()

class FlyBehaviour(ABC):
    @abstractclassmethod
    def fly(self):
        pass

class FlyHigh(FlyBehaviour):
    def fly():
        print('flying high')
        return

class FlyLow(FlyBehaviour):
    def fly():
        print('flying low')
        return

if __name__ == "__main__":
    fly_high_behaviour = FlyHigh()
    fly_low_behaviour = FlyLow()
    quack_loudly_behaviour = QuackLoudly()
    display_text_behaviour = DisplayText()
    mountain_duck = Duck(fly_high_behaviour, quack_loudly_behaviour,
                          display_text_behaviour)
    # behaviours can change
    mountain_duck.fly_behaviour = fly_low_behaviour

```

Interfaces

Have in mind that some languages like Java have the concept of interfaces (abstract classes that contain methods that have no code and need to be implemented by the class that inherited from that class). In python the abstract base class can be used to implement an interface (with a module called abc).

Notice that we ultimately work with the interface and not with the classes that implement that interface. This is the whole point. For example in the duck class, we work with an abstract fly_behaviour, not with a specific fly behaviour like flyHigh for example.

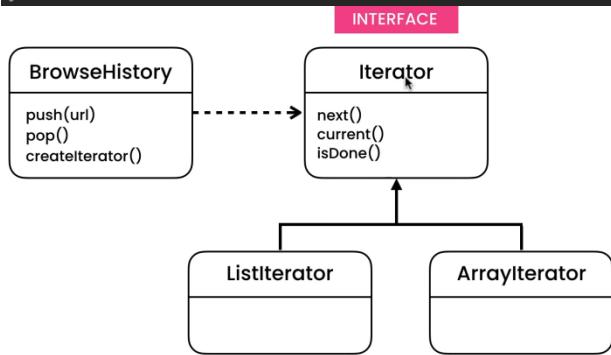
Iterator

```

public class Main {
    public static void main(String[] args) {
        var history = new BrowseHistory();
        history.push(url: "a");
        history.push(url: "b");
        history.push(url: "c");

        for (var i = 0; i < history.getUrls().size(); i++) {
            var url = history.getUrls().get(i);
            System.out.println(url);
        }
    }
}

```



Suppose we have implemented a history class (the `BrowseHistory` in this example) that holds the history states in a list. Then we use it in our code with methods that are specific to lists (push, size and get in this example). But what if we decide to change the list variable to another type? We would have to make changes to a lot of places in our code to change all the uses of list specific functions (push, size etc.). The solution is to use an iterator interface with three methods. Then we have specific implementations (list iterator). The `createIterator()` method returns a specific implementation instance.

The main function's loop, can be replaced with this. This way we can iterate over an iterator without caring about its internal representation.

```

while (iterator.hasNext()) {
    var url = iterator.current();
    System.out.println(url);
    iterator.next();
}

```

```

public class BrowseHistory {
    private List<String> urls = new ArrayList<>();

    public void push(String url) { urls.add(url); }

    public String pop() {
        var lastIndex = urls.size() - 1;
        var lastUrl = urls.get(lastIndex);
        urls.remove(lastUrl);

        return lastUrl;
    }

    public Iterator<String> createIterator() {
        return new ListIterator(history: this);
    }
}

```

The ListIterator is a nested class inside the BrowseHistory class (It's definition isn't visible in this picture). This way it can access browsehistory members. Then if we want to use another iterator we replace this class with another one and fix any errors appearing within the BrowseHistory class. But the main function remains without any errors.

```

public class ListIterator implements Iterator<String> {
    private BrowseHistory history;
    private int index;

    public ListIterator(BrowseHistory history) {
        this.history = history;
    }

    @Override
    public boolean hasNext() {
        return (index < history.urls.size());
    }

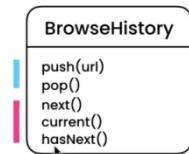
    @Override
    public String current() {
        return history.urls.get(index);
    }

    @Override
    public void next() {
        index++;
    }
}

```

The Iterator interface doesn't have any default logic, it only contains the empty abstract methods that the specific implementation have to implement. We need specific implementation to be able to create iterators of different types (list, array, object etc.).

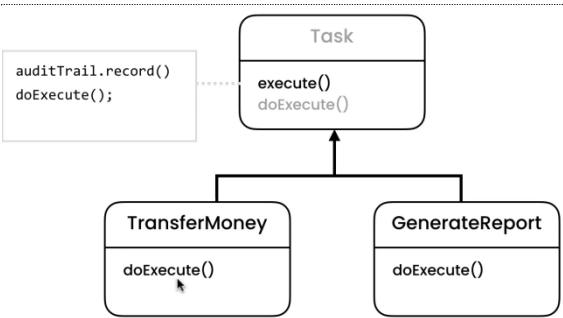
Notice that the iterator (it's specific implementation) gets the history object as an argument and knows how to iterate over it.



If we had all the methods in the browseHistory class then it would violate the single responsibility principle. It would be responsible for 1. history management and 2. for iteration. So we split it to two different classes (the BrowseHistory and Iterator respectively)

Template Method

A template is a sequence of methods execution. The body of the executed methods is defined in the implementation of the interface.



So Instead of this

It is presented with an example of an Audit trail: a system that traces the detailed transactions related to any item in an accounting record. Every time a user makes an action it would be recorded in an audit trail. The audit trail is represented by a class. In the example we have two actions, transfer money and generate report. Each task class has to define an execute() method that runs the audittrail record() method so that the action is recorded in the audittrail and runs the task's logic (doExecute) too. The audittrail is an instance of the Audittrail class and holds a list of executed actions.

The execute() method of the task interface defines a template, a sequence of methods execution. With this implementation when you are creating a new task, you only have to implement its actual logic (the doExecute method) and not the other things related to that task, that need to be executed too (here the other thing is the recording of the task in the audit trail)

Grey fonts are for abstract classes and methods.

```

Main.java x TransferMoneyTask.java x GenerateReportTask.java x Audit.java
package com.codewithmosh.template;

public class GenerateReportTask {
    private AuditTrail auditTrail;

    public GenerateReportTask(AuditTrail auditTrail) {
        this.auditTrail = auditTrail;
    }

    public void execute() {
        auditTrail.record();
    }

    System.out.println("Generate Report");
}

```

We simply do this for every new task

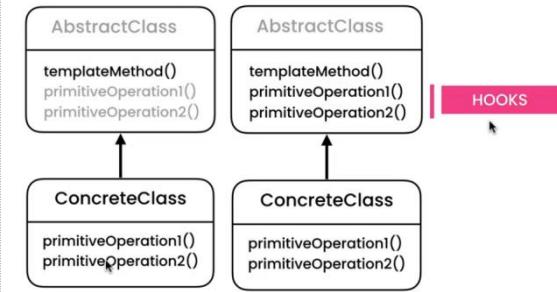
```

Main.java x TransferMoneyTask.java x Task.java x GenerateReportTask.java x Audit.java
package com.codewithmosh.template;

public class TransferMoneyTask extends Task {

    @Override
    public void doExecute() {
        System.out.println("Transfer Money");
    }
}

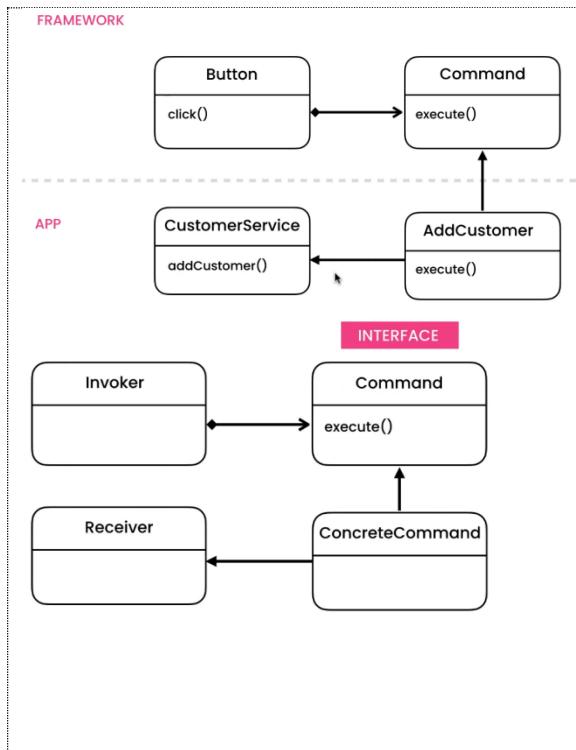
```



The concept is that you have a method of an abstract class (interface) which executes a few other abstract methods. This execution sequence is like a template. The implementations of this interface define the actual implementation of the methods of the template. A template of executing our class. Notice that the template methods could not be abstract but normal methods that the subclass can override if it needs to. These methods are called **hooks**, and this is a very common scheme in many frameworks. By overriding these hooks, you can modify the template.

Command

You create action objects for all actions of your application. This way you can create command sequences and undo functionality.



Assume we are building a gui framework (buttons etc.) that other developers can use to implement their functionality. We could have a button class. Each button has a label, and a click method that implements the logic that is executed when clicked. This should be implemented by the app developers not by us.

Once more, we use interfaces. The button receives a command instance (an implementation of an interface) with an execute method. On click, that method will be called. The app developer let's say wants to build addcustomer functionality on a customer service application. He builds an implementation of our command interface. In his implementation he calls the addCustomer method of his CustomerService class. Notice that the receiver (the customerService) is passed as argument to the command implementation, not the other way around. Then he can import the button class, and pass it his command implementation. On click a customer will be added.

We decouple the invoker from the receiver.

Since the concrete command implementations are classes, we can have instances of them, that can be stored in a list and in the database etc. So we can easily replay actions for example.

Composite Command

```

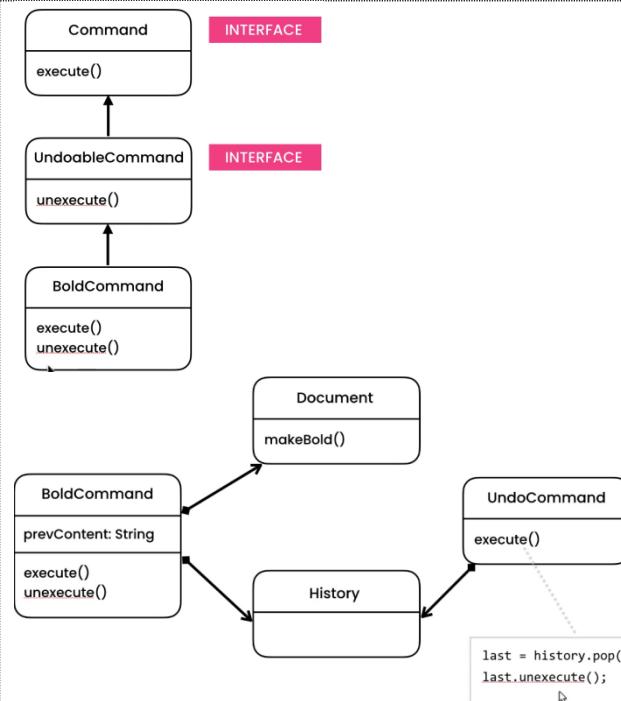
public class AddCustomerCommand implements Command {
    private CustomerService service;

    public AddCustomerCommand(CustomerService service) {
        this.service = service;
    }

    @Override
    public void execute() {
        service.addCustomer();
    }
}

public class Main {
    public static void main(String[] args) {
        var service = new CustomerService();
        var command = new AddCustomerCommand(service);
        var button = new Button(command);
        button.click();
    }
}

```



```

package com.codewithmosh.command.editor;

public class UndoCommand implements Command {
    private History history;

    public UndoCommand(History history) {
        this.history = history;
    }

    @Override
    public void execute() {
        if (history.size() > 0)
            history.pop().unexecute();
    }
}

Main.java
var document = new HtmlDocument();
document.setContent("Hello World");

var boldCommand = new BoldCommand(document, history);
boldCommand.execute();
System.out.println(document.getContent());

var undoCommand = new UndoCommand(history);
undoCommand.execute();
System.out.println(document.getContent());

```

For example, we can create a composite command (which is also an implementation of the command interface) that gets a list of command implementations and executes them in the execute method.

```

public class CompositeCommand implements Command {
    private List<Command> commands = new ArrayList<>();

    public void add(Command command) {
        commands.add(command);
    }

    @Override
    public void execute() {
        for (var command : commands)
            command.execute();
    }
}

```

Undoable commands

The command pattern can also be used to implement undo functionality. We make any action we want, undoable. We do this by creating a new interface (UndoableCommand) that extends the command interface. It has the unexecute method. We don't put this method in the original command interface because not all actions are undoable. The boldCommand (makes the fonts bold) is the concrete implementation of the undoable interface. The receiver is a distinct class (the Document in this case). it knows how to make its content bold.

Notice that we store the previous content in the command implementation. It will be used by the unexecute method to bring the content to its original state. The prev state is command specific. For example in this case is the text content without the bold marks. In case of a resize action, it would be the dimensions of the page before the resize. All undoable commands can push themselves to a global history object. The undo command is a command itself, which implements the command interface.

Bold Command execute()

```

prevContent = doc.getContent();
document.makeBold();
history.push(this);

```

Bold Command unexecute()

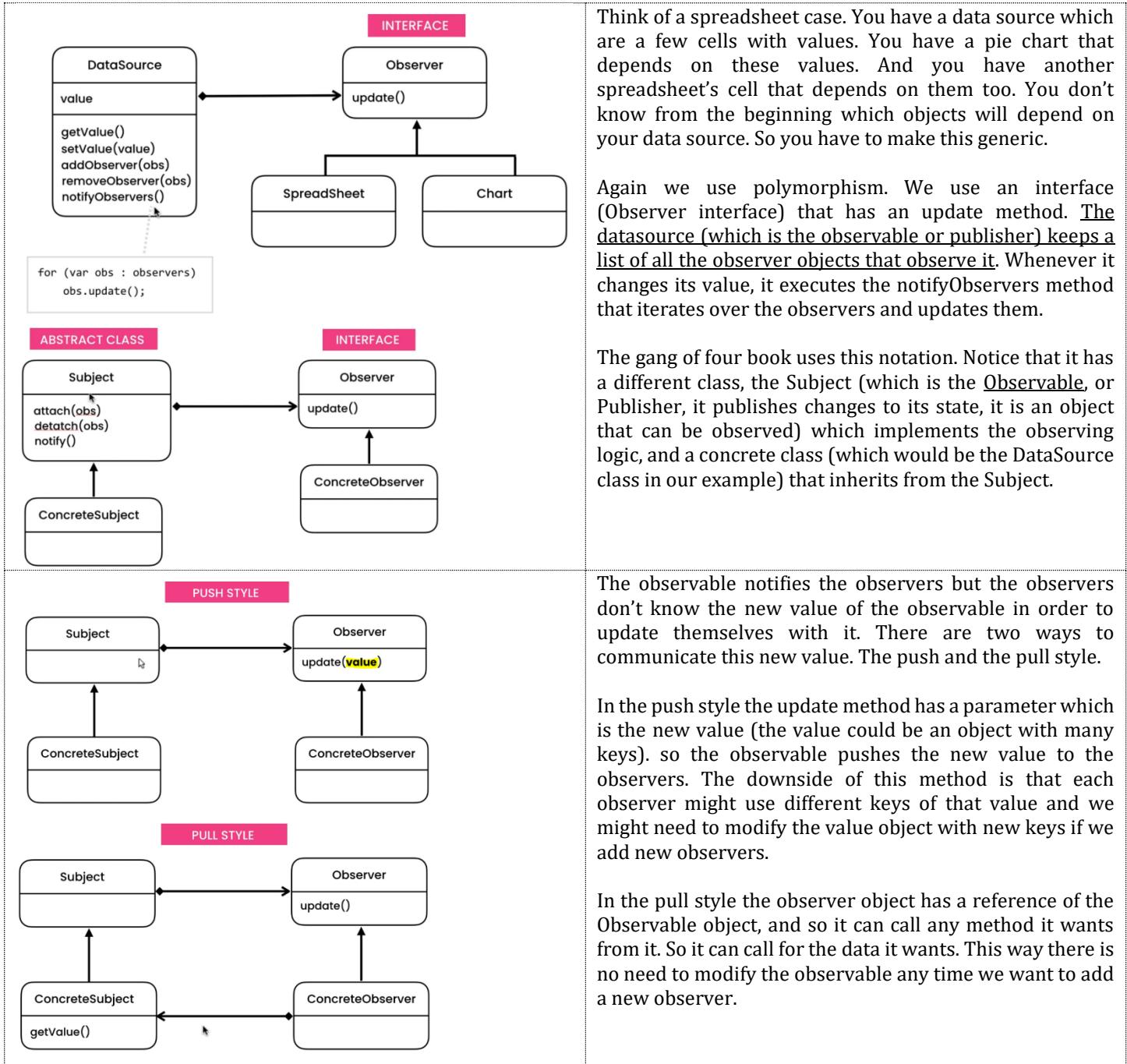
```

doc.setContent(prevContent)

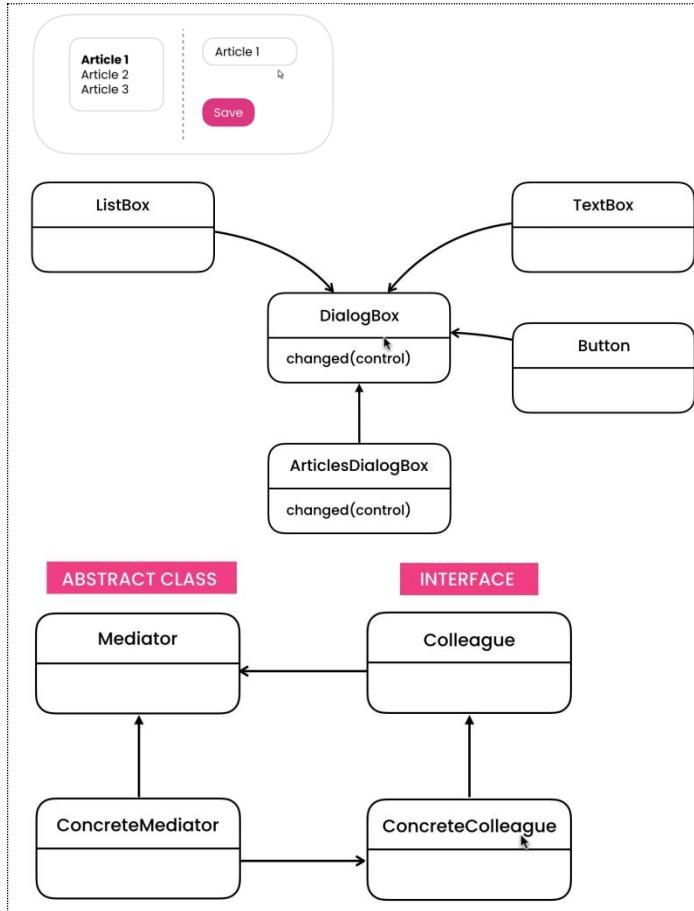
```

See the differences with the memento pattern in the memento chapter.

Observer (Publish Subscribe)



Mediator



Suppose that we want to create a dialog box window that has three sections. An articles list from which you can select one. A title field which shows the title of the selected article (we can edit it). And a button that saves the title when pressed. If no article is selected, no title is shown and the button is inactive.

There is a relationship between the three sections (controls). we want to implement this relationship in an extensible way. This is what the mediator pattern solves. It creates a mediator class (the ArticlesDialogBox in this case) that handles the relationship logic while the controls themselves don't have to know about other controls that are related to them.

Since there might be some common logic for all the dialog boxes of our application, we implement the DialogBox as an abstract class. So each dialogbox that handles its own controls, would be a distinct implementation of the abstract dialogBox.

The most convenient implementation of the Mediator pattern is to make the Controls observables, and the specific dialog box an observer of them. In any case, the mediator class is the one that handles all the relationship logic.

First Implementation

The UIControl class

```

package com.codewithmosh.mediator;

public class UIControl {
    protected DialogBox owner;

    public UIControl(DialogBox owner) {
        this.owner = owner;
    }
}

```

The ListBox class

Every control (the button, the list, the text box) are children of an abstract UIControl class. The UIControl class has an owner field which is a specific dialogBox instance. When you will create a specific ui control like the Button, you will initialize it with a specific DialogBox (which will be the button's owner). When the button's content changes, it will call it's owners changed() method and will pass itself to it. This way the owner (which is the specific dialog box) will know which of it's controls has changed and will apply any logic with it, using an if else statement to get which of it's controls has changed.

The ArticlesDialogBox class

```

package com.codewithmosh.mediator;

public class ArticlesDialogBox extends DialogBox {
    private ListBox articlesListBox = new ListBox(owner: this);
    private TextBox titleTextBox = new TextBox(owner: this);
    private Button saveButton = new Button(owner: this);
}

```

```

package com.codewithmosh.mediator;

public class ListBox extends UIControl {
    private String selection;

    public ListBox(DialogBox owner) {
        super(owner);
    }

    public String getSelection() {
        return selection;
    }

    public void setSelection(String selection) {
        this.selection = selection;
        owner.changed(control: this);
    }
}

```

```

@Override
public void changed(UIControl control) {
    if (control == articlesListBox)
        articleSelected();
    else if (control == titleTextBox)
        titleChanged();
}

private void titleChanged() {
    var content = titleTextBox.getContent();
    var isEmpty = (content == null || content.isEmpty());
    saveButton.setEnabled(!isEmpty);
}

private void articleSelected() {
    titleTextBox.setContent(articlesListBox.getSelection());
}

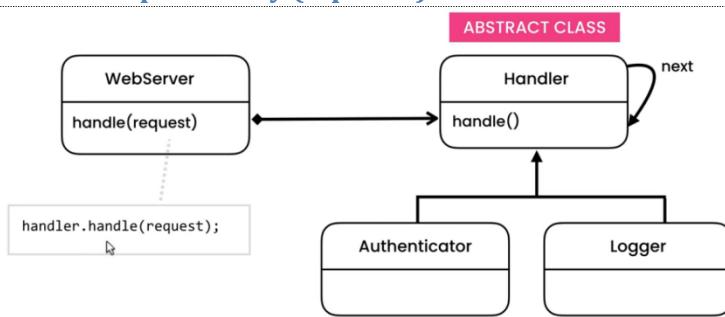
```

Implementation using the Observer pattern

In this implementation there is no need for the UIControl to have an owner field. The dialogbox would be UIControl's observer, and the uicontrol will not have to know about it.

Notice: In this implementation the update() method of the observer (of the articles dialog box) can be called handler() as it handles the event of a control change. The observer interface can be called EventHandler (an object that handles an event that happened somewhere else). The attach() method of the observable (of the controls) can be renamed to addEventListeners. Notify to notifyEventHandlers. It is a naming convention thing, used by many frameworks.

Chain of responsibility (Pipeline)



We want to implement a pipeline, where one action is followed by another one. We don't want to hard code the order of actions in the pipeline, nor to the pipenodes themselves. We want to be able to control the order externally easily.

So we implement it with a linked list actually. We use a Handler interface with a next variable that is a Handler itself. Each handler knows its next one. We start the execution of the pipeline by passing the data (the request in this case) from the first handler. Then it passes it to the next one and so on. (I made a repl test case with this example).

Notice that the Pipeline is a new object, which has a simple reference to the first Pipenode of the pipeline and when you execute the pipeline, it executes the first pipenode.

```

class Authenticator(PipeNode):
    def do_execute(self, data):
        request = data
        if request.username == 'gimli' and request.password == 'dwarf':
            print("authenticate")
        else:
            return "Unauthenticated"

```

```

main.py x
1  from pipeline import Compressor, Logger, Aut
2
3  def main():
4      # authenticate -> log -> compress
5      compressor = Compressor(None)
6      logger = Logger(compressor)
7      authenticator = Authenticator(logger)
8      pipeline = Pipeline(authenticator)
9
10     request = HttpRequest(password='dwarf')
11     pipeline.execute(request)
12
13 if __name__ == '__main__':
14     main()
15
16

```

```

class Pipeline():
    def __init__(self, pipenode):
        self.pipenode = pipenode

    def execute(self, data):
        self.pipenode.execute(data)

```

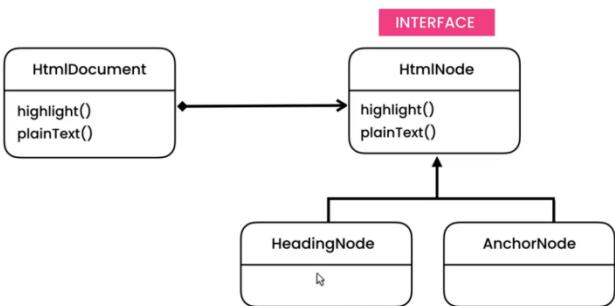
```

pipeline.py x
1  from abc import ABC, abstractmethod
2
3  class PipeNode(ABC):
4      def __init__(self, next):
5          # next is a PipeNode
6          self.next = next
7
8      def execute(self, data):
9          if self.do_execute(data) != 'Unauthenticated':
10              self.execute_next(data)
11
12      # the PipeNode logic is implemented here
13      @abstractmethod
14      def do_execute(data):
15          pass
16
17      def execute_next(self, data):
18          if self.has_next():
19              self.next.execute(data)
20
21      def has_next(self):
22          return self.next

```

Visitor

The naive implementation



```

Main.java x 1 HtmlNode.java x 2 HtmlDocument.java x 3 HeadingNode.java x
private List<HtmlNode> nodes = new ArrayList<>();

public void add(HtmlNode node) {
    nodes.add(node);
}

public void highlight() {
    for (var node : nodes)
        node.highlight();
}

```

The visitor pattern

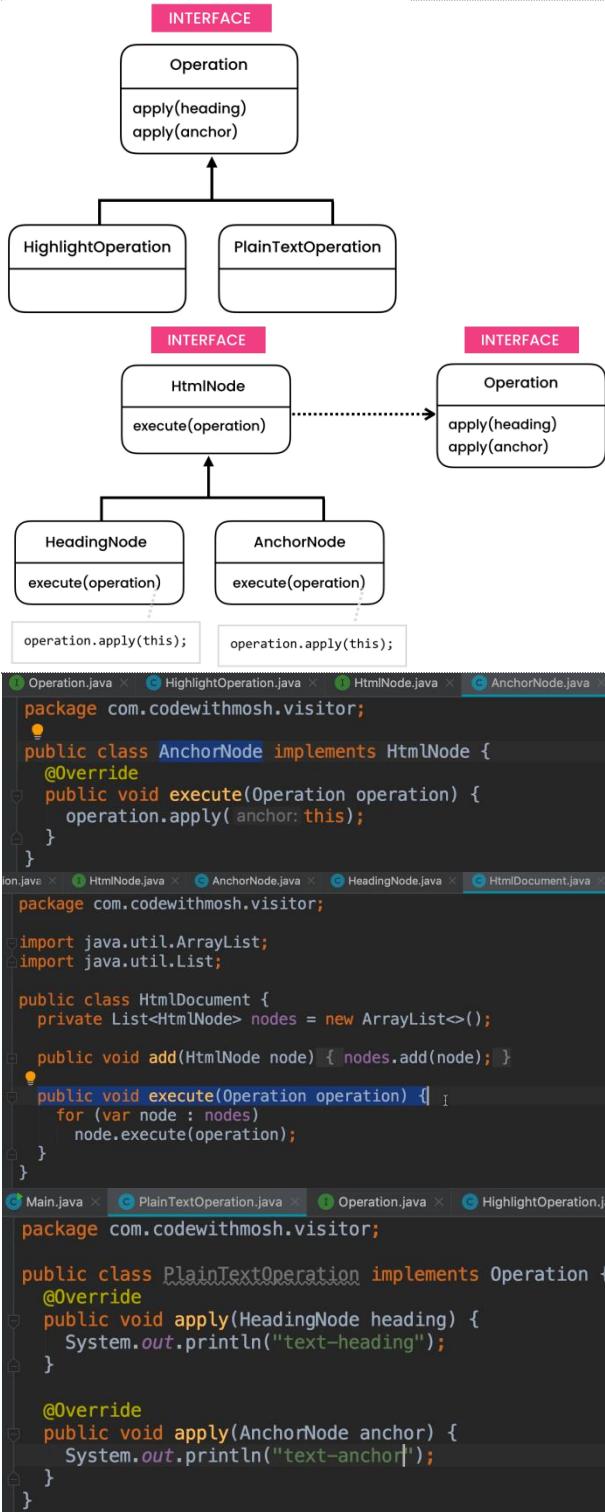
It allows to add new operations to an object structure without modifying it.

In the naive implementation whenever we want to add new functionality to the htmlNodes, for example the plainText method which returns the inner text of the node, we must add an abstract plainText method to the interface, then write it in all implementations and create it in the htmlDocument which iterates over all added html nodes and executes their plainText method. This is not an easily extensible implementation.

Method overloading

A class can have more than two methods with the same name but different signatures (for example different parameters). Notice: in python you can implement method overloading with a library like “typing”.

We use method overloading to solve this problem. We create an operation interface which has two “apply” abstract methods each receiving a specific html node. Then each specific operation (highlight, innerText etc.) has its own implementation of the interface and implements all versions of the apply method (one method for each node where the method contains the code that applies this specific operation to the specific html node). this way all the logic of that operation and how it applies to the different nodes, is in one place. Then each html node



implementation has an `execute` method that gets an `operation` and runs its `apply` method passing itself. This way we don't add new methods to the html nodes implementation, but we keep the logic of an operation and how it is applied to all elements, in one place.

We use this approach whenever the object structure (the number of objects to which the operations are applied) doesn't change often, but we want to be able to frequently add additional methods to it. For example in this case, there is a fixed amount of html nodes that exist, let's say 20 nodes, so the operation interface would have to have the `apply` method 20 times. But we only touch the interface code once, to create it. There are no new html nodes all the time so we rarely have to change the Operation interface adding more `apply` methods to it. On the contrary there are many operations that can be applied to an html node, and we can implement them easily by adding one more Operation implementation. This follows the open closed principle. You can just add one more implementation class to add new functionality.

You don't have to modify the specific html node implementation with a new method since it has an extensibility point (`operation.apply(this)`). This, will execute only the `apply` method that expects this specific object.

The `htmlDocument` has also an extensibility point. So we can pass new operations to it without modifying it. That's the beauty of the visitor pattern.

```

public class Main {
    public static void main(String[] args) {
        var document = new HtmlDocument();
        document.add(new HeadingNode());
        document.add(new AnchorNode());
        document.execute(new HighlightOperation());
    }
}
  
```

The go4 book named the `Operation` interface "Visitor" and the `HtmlNode` "Element" while the `execute(operation)` method "accept(visitor)".

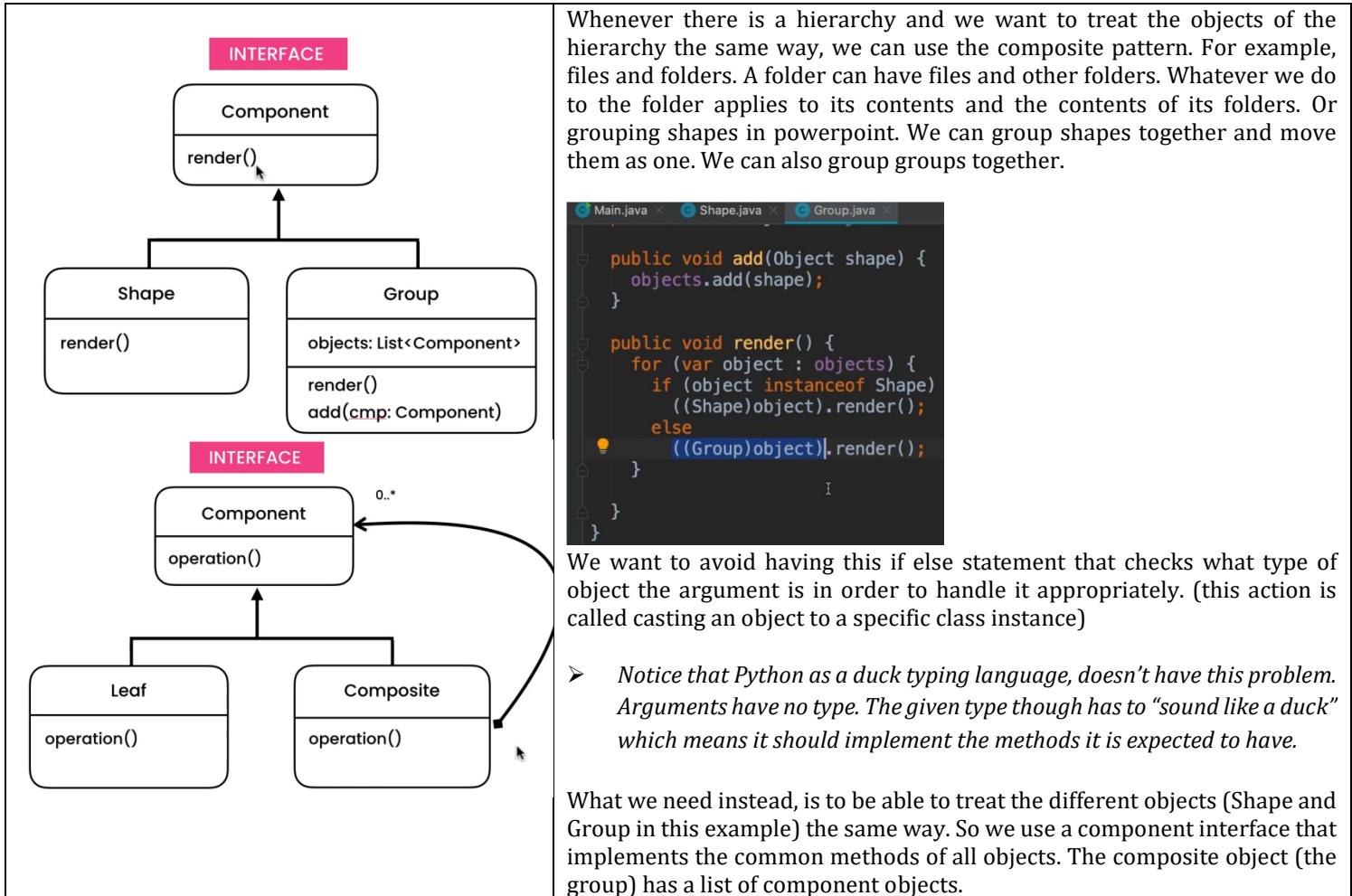
-- Structural --

patterns that define the relationships between objects. Have in mind Composite, Adapter, Proxy.

Notice that Python as a duck typing language, doesn't face some of the issues described. Arguments have no type. The given type though must "sound like a duck" which means it should implement the methods it is expected to have.

Composite

Promotes composition. Similar to the strategy pattern.



Adapter

We use it to convert an interface of an object to a different form (like real world power adapters).

Imagine that we build an application that applies filters to images. We build a **Filter** interface with an **apply** method. Then we can make various filter implementations. Then in our code we can pass an image through a **Filter** object. Now imagine that we want to use a third party library that has its own filters. These filters are not implementations of our **Filter** interface so our main code can't accept them (type inconsistency). To solve this we have to somehow convert the interface of the third party filters to match our own interface.

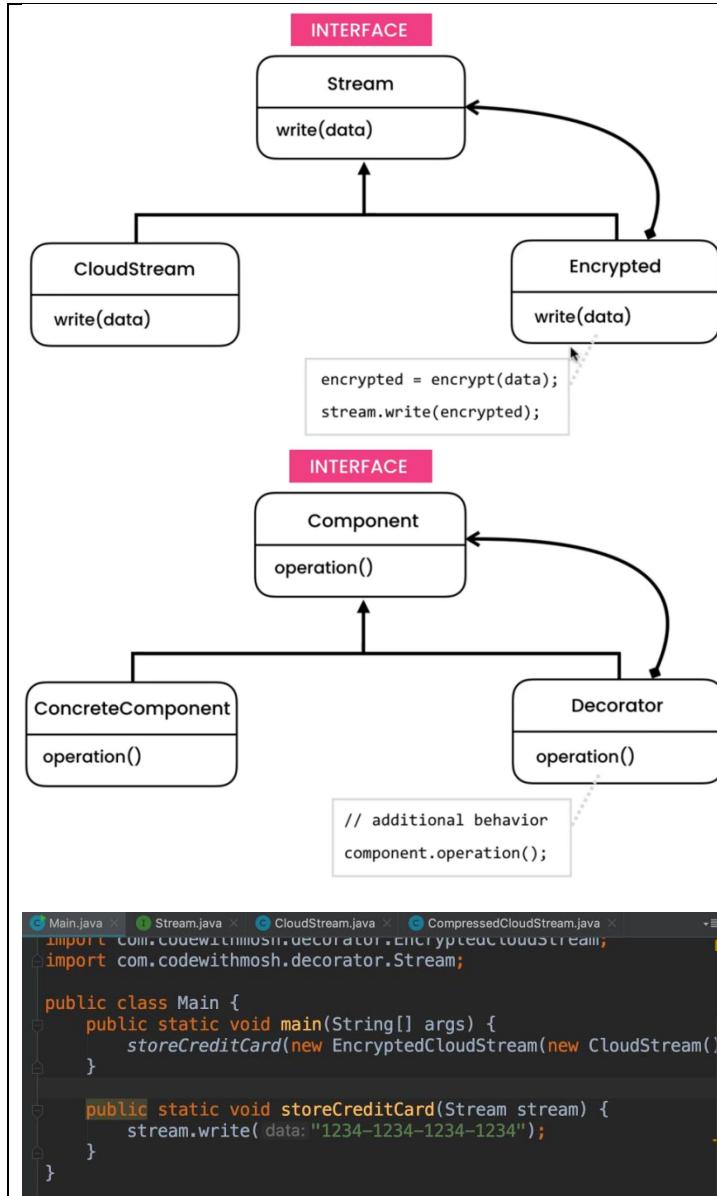
The solution is to create a new **Filter** implementation (as if we create a new of our filters) that implements our interface, but it is composed of the third party object, which means that it calls the third party methods when its **apply** method is called. So we actually convert the interface of the third party using an adapter.

- In Python as a duck typing language, arguments have no type. The given type has to "sound like a duck" which means it should implement the methods it is expected to have. This means that the argument can be a **Filter** instance or whatever other class instance. It just have to implement the methods that are being called from it. Notice though that this pattern can still be useful in python since the 3rd party library objects could have different method names than our own, so we would need to wrap them in our own objects and call them from there.

From other source: Adapter is very commonly used, Bridge is quite rare.

Decorator

We decorate an object with additional behavior.



Suppose you want to write a class (CloudStream) that reads and writes data in the cloud. It has a write method. In cases where the data is sensitive you want to encrypt it before writing to the cloud. For other data you might want to compress before writing, or compress and encrypt. The naive approach would be to create one child class (child of CloudStream) for each operation and override its write method with additional logic. This though would mean that for every new operation you would have to create new classes, not only for the new operation but for combinations of operations. This is not maintainable.

The solution is to use composition instead of inheritance. You create a Stream interface. Then you have one stream implementation for each operation. These implementations (except from the CloudStream class) have a variable self.stream = stream (they are composed of the Stream object) and they can use its methods. So the write method of each class, will apply its operation logic (like encryption) and then it will pass the operated data to its stream object to handle.

The Encrypted class is a decorator and the CloudStream class is the decorated class, because it performs some arbitrary logic (decorating) before executing the decorated logic.

To use it you pass the decorated stream as argument to the decorator stream: EncryptedCloudStream(Cloudstream)

It reminds me of the pipeline structure but this structure is I guess not suitable for long chains of operations. It is mostly for adding some additional operations that are performed from time to time, to an object that you use very often.

Here we have one specific object (the CloudStream) that needs to be decorated with additional functionality. In pipeline all objects are equal and just have a reference to the next one.

Facade

It's actually an implementation of the abstraction OOP principle.

```

import com.codewithmosh.facade.Message;
import com.codewithmosh.facade.NotificationServer;

public class Main {
    public static void main(String[] args) {
        var server = new NotificationServer();
        var connection = server.connect(ipAddress: "ip");
        var authToken = server.authenticate(appID: "appID", key: "key");
        var message = new Message(content: "Hello World");
        server.send(authToken, message, target: "target");
        connection.disconnect();
    }
}

```

We have implemented a notificationServer class to send push notifications to our app clients. The problem is that every time we want to send a notification we have to repeat this whole sequence of actions (connect, get auth token, send the message, disconnect)

The other problem is that our main code that sends the messages, depends on 4 different classes (notificationServer, Message, AuthToken, Connection) to complete this action.

The solution is to create a new class (NotificationService) that has a send method that executes the sequence of actions and is dependent on the 4 classes. Our main code now only has to depend on the NotificationService (which is the facade or the front of its underlying code)

Flyweight

Reduce the amount of memory used by our objects by using a reference (for example an id) to the heavy object instead of the object itself.

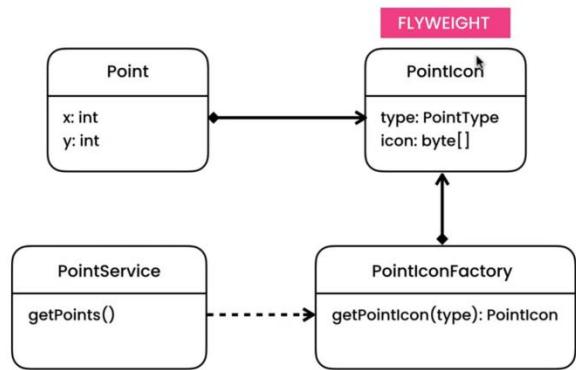
```

public class Point {
    private int x; // 4 bytes
    private int y; // 4 bytes
    private PointType type; // 4 bytes
    private byte[] icon; // 20 KB -> 20 MB

    public Point(int x, int y, PointType type, byte[] icon) {
        this.x = x;
        this.y = y;
        this.type = type;
        this.icon = icon;
    }

    public void draw() {
        System.out.printf("%s at (%d, %d)", type, x, y);
    }
}

```



For example in case of a google maps like app where we have points and each point has x,y coordinates, a type (restaurant, hotel etc.) and an image (stored as a byte array in this example). if we have 100 hotels the 100 points can take a lot of memory due to the size of the image. 100 images of a hotel. Instead we separate the type and image in a different class (PointIcon) and use a reference to it in the point class. Any time we want to create a new image we do it through a factory class that ensures that it will only create the image if it doesn't already exist. If it exists it will return it (from some kind of memory or cache). The flyweight object (the pointIcon in this case) is the object that can be referenced.

```

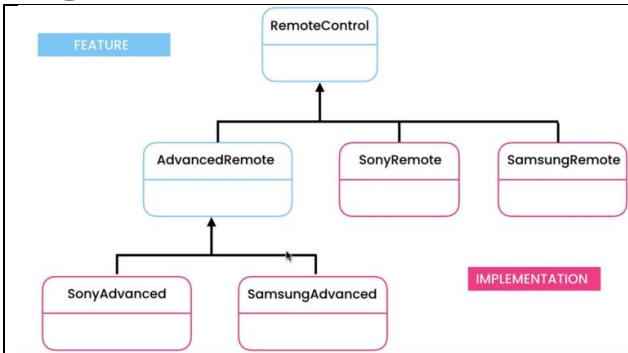
import java.util.HashMap;
import java.util.Map;

public class PointIconFactory {
    private Map<PointType, PointIcon> icons = new HashMap<>();

    public PointIcon getPointIcon(PointType type) {
        if (!icons.containsKey(type)) {
            var icon = new PointIcon(type, icon: null);
            icons.put(type, icon);
        }
        return icons.get(type);
    }
}

```

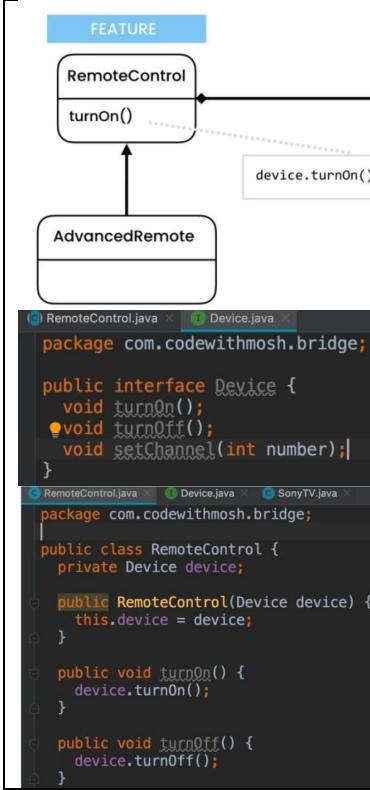
Bridge



Suppose that we are building a remote control for various devices. Let's say we have an abstract RemoteControl class with two methods (turn on and off) and an implementation of it called, advancedRemoteControl with more functionality (change channel). We must create a remoteControl and an Advanced one for any tv model we want to support. This is not an extensible approach. We have a hierarchy of two different dimensions. Feature and implementation. We need to split these two hierarchy dimensions and let them grow independently.

What we do is to create a Device interface with all the common methods. Then each tv model can be an implementation of that interface.

The remoteControl is now not an interface but a standard class that has a device variable (this.device, is composed from a device). so we can pass it a device and have it execute the device's methods.



With the bridge pattern we split this hierarchy to two branches. The features hierarchy (remote and its child advanced) and the devices hierarchy (device interface and its children)

The composition relationship between the remoteControl and the Device classes is the bridge between the two hierarchies.

Map in java is like a python dictionary

```

Main.java  RemoteControl.java  AdvancedRemoteControl.java  Device.java

package com.codewithmosh;

import com.codewithmosh.bridge.RemoteControl;
import com.codewithmosh.bridge.SonyTV;

public class Main {
    public static void main(String[] args) {
        var remoteControl = new RemoteControl(new SonyTV());
        remoteControl.turnOn();
    }
}

```

Proxy

We create a proxy (or agent) for a real object and talk to the real object through the proxy. The benefit is that in this proxy we can do some arbitrary logic, like logging, caching, access control etc.

```

Main.java  Ebook.java

public void show(String fileName) {
    this.fileName = fileName;
    load();
}

private void load() {
    System.out.println("Loading the ebook " + fileName);
}

public void show() {
    System.out.println("Showing the ebook " + fileName);
}

public String getFileName() {
    return fileName;
}

```

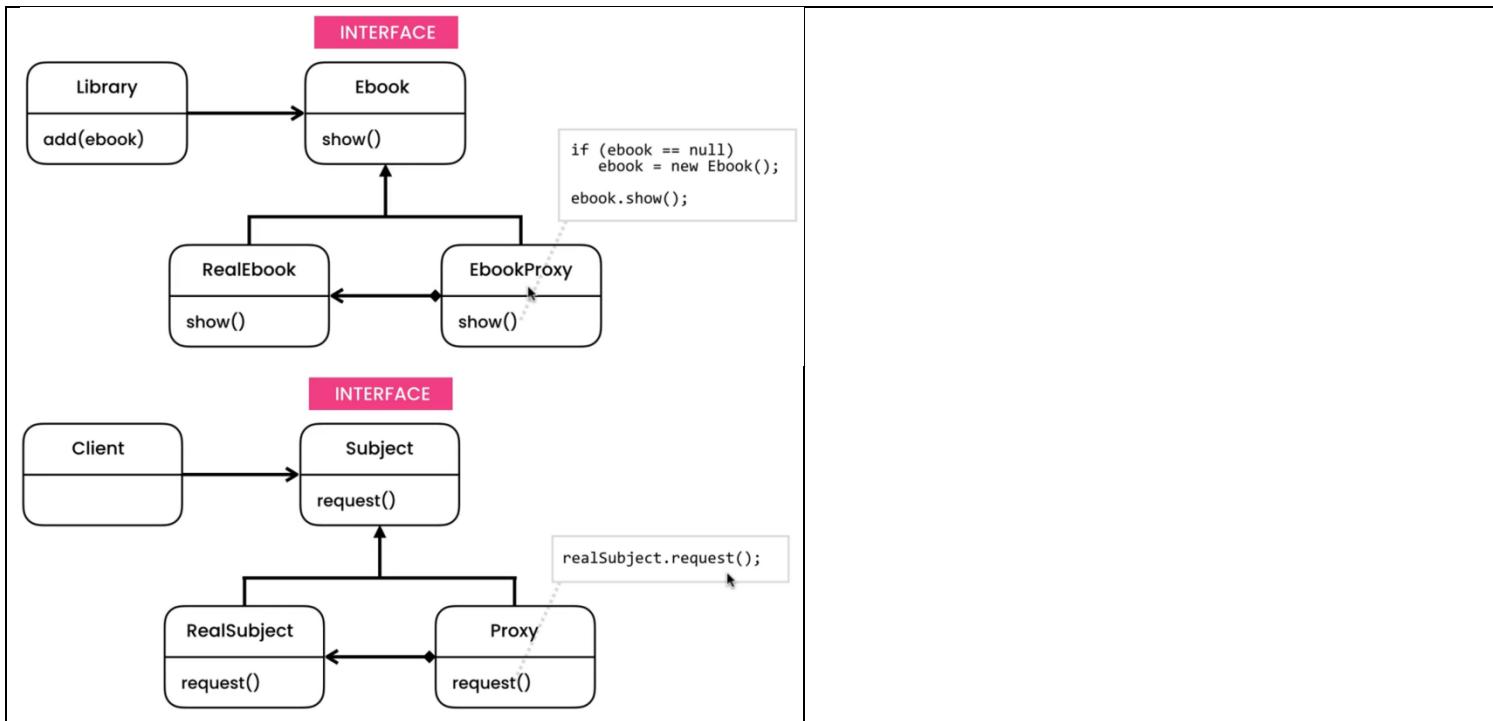
This is an example of using the proxy for implementing Lazy initialization.

Suppose we are building an ebook library app. We have an ebook and a library class. Whenever an ebook instance is created, it loads the ebook file in memory (it runs the load method on initialization). this might cause our app to use a lot of memory without a reason, since we might only want to ultimately show one of the loaded books.

So we implement a Proxy class (the EbookProxy which is an implementation of an Ebook interface. The real ebook that doesn't do lazy initialization is also one) and we use the EbookProxy in our code. The proxy doesn't load the ebook file in memory when it is created, but when it is to be shown. (django uses lazy initialization for its querysets for example). the proxy is composed of the real object (self.ebook = ebook). notice that we don't initialize it in the object creation, but only on the show method.

We can add more proxies for additional functionality (logging, access control etc.) in this implementation. It follows the open closed principle. I guess that we can combine more than one proxies (passing one to the other) but I don't know if this is a good practice.

Essentially the proxy is identical with a realSubject (since it implements the same interface) but it delegates its method execution to the real subject.

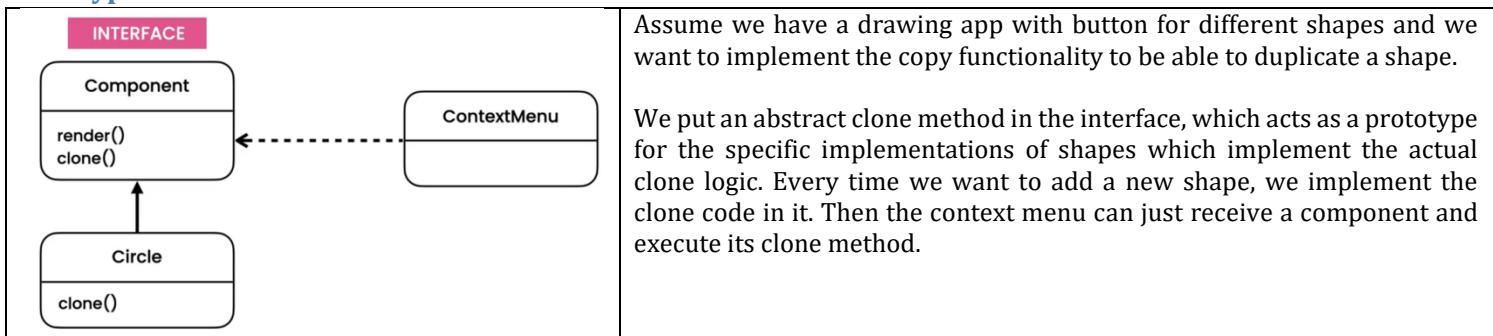


--- Creational ---

Patterns that deal with different ways to create objects.

Have in mind Abstract factory.

Prototype



Singleton

<pre> class ConfigManager { get() set() -ConfigManager() -instance getInstance() } </pre>	<p>There are cases in which we should assure that only one instance of a class is created and used, for example for storing our application's settings. Here we have a configManager class that is responsible for creating the app's settings. We make the constructor private so the new command (to create a new configManager) can't be used outside of this class. We also have an instance field which is responsible for storing the actual single instance of the class. It is private and static (underlined). whenever we want to get this instance we use the getInstance method (which is also static, because static fields are only visible to static methods)</p>
---	--

Factory

We use it to defer the creation of an object to subclasses

```

package com.codewithmosh.factory.matcha;

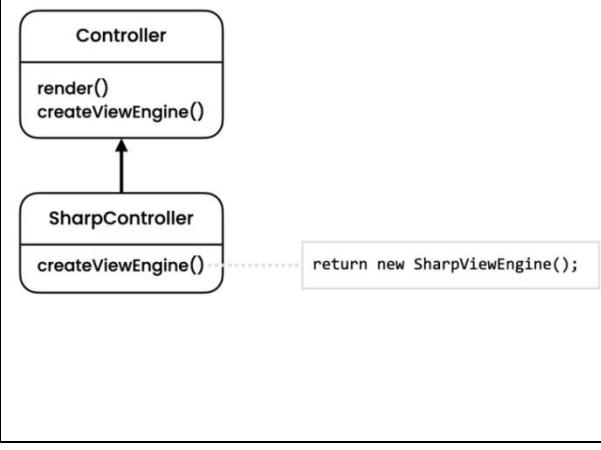
import java.util.Map;

public class Controller {
    public void render(String viewName, Map<String, Object> context) {
        var html = engine.render(viewName, context);
        System.out.println(html);
    }
}

import java.util.HashMap;
import java.util.Map;

public class ProductsController extends Controller {
    public void listProducts() {
        // Get products from a database
        Map<String, Object> context = new HashMap<>();
        // context.put(products)
        render(viewName: "products.html", context, new_);
    }
}

```



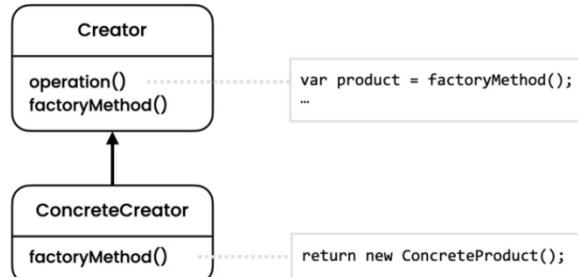
Assume we are building a web framework named Matcha for other developers to use. We have a MatchaViewEngine class (that implements a ViewEngine interface) that has a render method that gets a view name and a context and knows how to create an html page from them. We also create a Controller class that gets a specific engine as an argument and uses its render method.

Developers could extend our controller class to build their own controllers that use the render method. The controller could get any view engine to render the page.

The problem is though that now developers need to pass the specific engine every time they use our render method. This is not convenient.

The solution is to add a createViewEngine method inside the Controller class and have the render method of the class to call that method to get the engine. Then developers can override the createViewEngine class in their Controller implementations if they want to use a different engine.

In this case the createViewEngine method is a factory method of engine objects which defers the creation of the objects to subclasses (if the createViewEngine is an abstract method).



Abstract factory

It provides an interface for creating families of related objects

```

Project
  out
  src
    com.codewithmosh
      abstractFactory
        ant
        app
        material
          Button
          TextBox
        Theme
        Widget
      ContactForm.java

import com.codewithmosh.abstractFactory.ant.AntTextBox;
import com.codewithmosh.abstractFactory.material.MaterialButton;
import com.codewithmosh.abstractFactory.material.MaterialTextBox;

public class ContactForm {
    public void render(Theme theme) {
        if (theme == Theme.ANT) {
            new AntTextBox().render();
            new AntButton().render();
        } else if (theme == Theme.MATERIAL) {
            new MaterialTextBox().render();
            new MaterialButton().render();
        }
    }
}

```

Assume we want to build an ui framework that has two widgets, a button and a textbox. We also have two themes (material and ant) and the widgets change according to which theme is selected. So, we have one family of widgets for the material ui theme and another one for the ant ui theme. The naive way to implement a contact form that rendered these widgets would be with an if else statement which violates the open closed principle.

The solution is to use a widgetFactory interface which has two methods. One for creating a button object and one for a textbox object. Then we can have various implementations of this abstract widgetFactory. A material and an ant factory implementation in our case. Each would implement the two methods differently and would return the factory specific widgets. The widgetFactory in this example is an abstract factory that has two factory methods.

Then instead of passing a theme enum to the contactForm you pass it a specific widgetFactory.

```

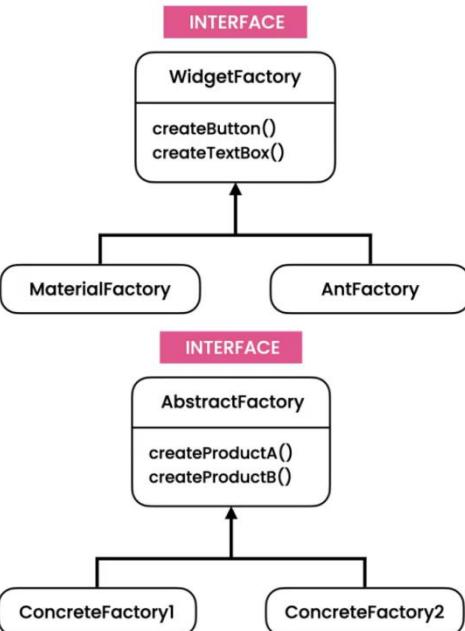
1 WidgetFactory.java x 2 MaterialWidgetFactory.java x 3 AntWidgetFactory.java x
package com.codewithmosh.abstractFactory.app;

import com.codewithmosh.abstractFactory.WidgetFactory;

public class ContactForm {
    public void render(WidgetFactory factory) {
        factory.createTextBox().render();
        factory.createButton().render();
    }
}

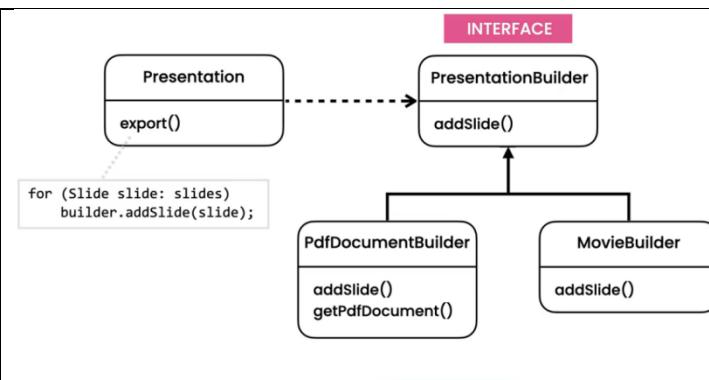
Main.java x 1 WidgetFactory.java x 2 MaterialWidgetFactory.java x 3 AntWidgetFactory.java x
public class Main {
    public static void main(String[] args) {
        new ContactForm().render(new MaterialWidgetFactory());
    }
}

```



Builder

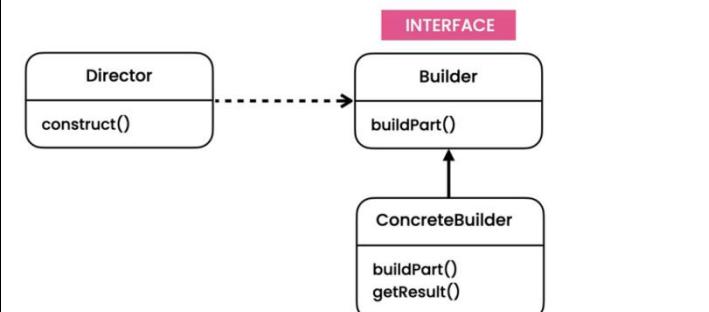
It is used to separate the construction of an object from its representation.



Assume we work on a powerpoint like app, and we want to build the feature of exporting our current presentation in different formats like pdf, image, movie etc. Again, instead of using an if else statement to check the selected presentation format, we use the builder pattern.

We actually create a `PresentationBuilder` interface with an `addSlide` method. We create a specific implementation for each output format that implements its logic in the `addSlice` method. Then in the main code, (the `Presentation` class) we work against the abstract `PresentationBuilder` interface.

(In this case construction is the exporting logic and its representation is the presentation format)



Have in mind that the `getResult` method (`getPdfDocument` in our case) is also necessary since the `export` method here, returns void and not a specific presentation builder. We need to call its method in order to get it.

```

1 PresentationBuilder.java x 2 PdfDocumentBuilder.java x 3 MovieBuilder.java x 4 Presentation.java x
import java.util.ArrayList;
import java.util.List;

public class Presentation {
    private List<Slide> slides = new ArrayList<>();

    public void addSlide(Slide slide) { slides.add(slide); }

    public void export(PresentationBuilder builder) {
        builder.addSlide(new Slide(text:"Copyright"));
        for (Slide slide: slides)
            builder.addSlide(slide);
    }
}

```

Misc

Functions whose output type is the same as its input type are really easy to compose together

Data structures

A data structure is a collection of data values, the relationships among them and the operations that can be applied to them.

Big O notation

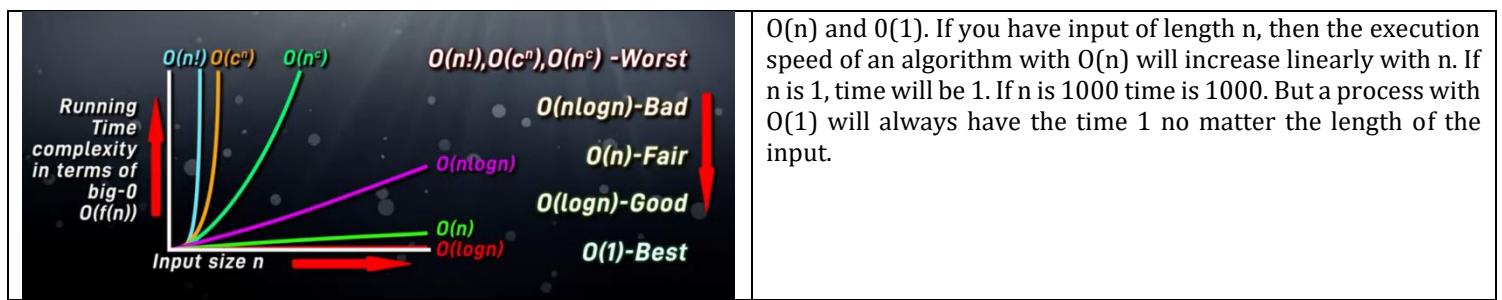
Big O notation is a measuring standard that shows **how much an algorithm is affected (in terms of execution speed and memory usage) by the size of its input**. We care about measuring how an algorithm is performing as the size of the input increases.

Time complexity: how much computation is needed

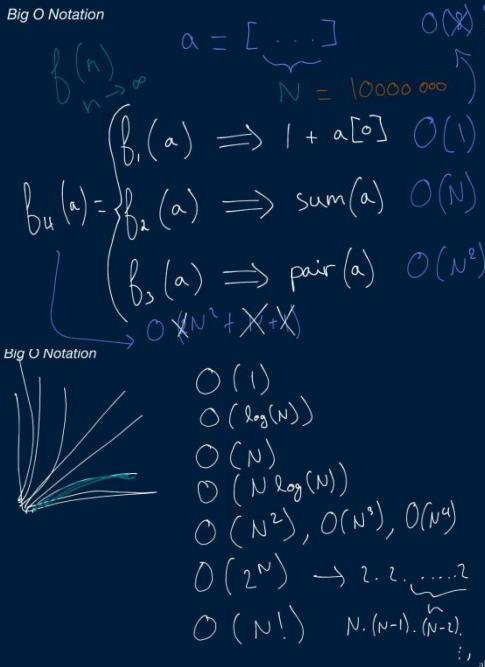
Space complexity: How much memory is needed

Memory

- It consists of a finite number of memory slots. A memory slot has its own memory address. A memory slot can contain 8 bits.
- Integers are fixed width data, meaning that they are stored as 8, 16, 32 or 64 bits. In C a variable int is a 32 bit integer. A long is a 64 bit integer. Fixed width data must be stored box to box, meaning in consecutive memory slots, one next to the other. So for a 64 bit integer you need 8 free consecutive memory slots.
- Any type of information can be transformed to a base 2 format and stored as bits into memory slots. For example strings can be mapped to numbers and numbers can be represented by base 2 numbers that can be stored in memory. One such mapping for strings is the ASCII which maps strings (letters) to numbers.
- To store a list you need box to box slots. The contents of the list must be stored box to box.
- Memory uses the concept of pointers. You can avoid storing a particular data value in a memory slot, and instead store the memory address of another memory slot that contains that data value. This is a pointer.
- Accessing a memory slot given its memory address is the most basic operation on a memory, and is a very fast operation.



Asymptotic analysis is the analysis of the efficiency of a function as its input size increases to infinity. $f(n)_{n \rightarrow \infty}$



The input is an array of length n.

- The first function has a constant time complexity or in other words $O(1)$. No matter the size of the array the speed of execution would be the same since it always accesses two items from the memory (integer 1 and the array value) and adds them. If these are 32 bit integers then reading each integer from memory requires 4 elementary operations (accessing 4 memory slots, the 4 bytes that constitute the integer). So in this example we have at least 8 elementary operations. So maybe we should say that the function is $O(8)$? No, we don't do that. As long as the number of elementary operations doesn't depend on the size of the input then we say that the function has a constant time complexity and we symbolize this with $O(1)$. So no matter the integer type (32 bits, 64 bits etc) the time and space complexity would be $O(1)$.
- The second function has to iterate over the whole array so its time complexity is linear $O(n)$
- The third function has two nested loops. For each element of the array, it iterates over the whole array. Iterating over the whole array requires n iterations, and it has to do this n times. So the time complexity is $O(n^2)$.

- Imagine that we have also a fourth function that requires the execution of the first, the second and two times the third one. We could say that its time complexity would be $2*O(n^2) + O(n) + O(1)$. But in asymptotic analysis the most important factor is the one that determines the efficiency of a function. So the time complexity of the fourth function would be $O(n^2)$. the reason is that as n gets bigger, the other terms become insignificant. Also the 2 is a constant factor and constant factors are not significant too.

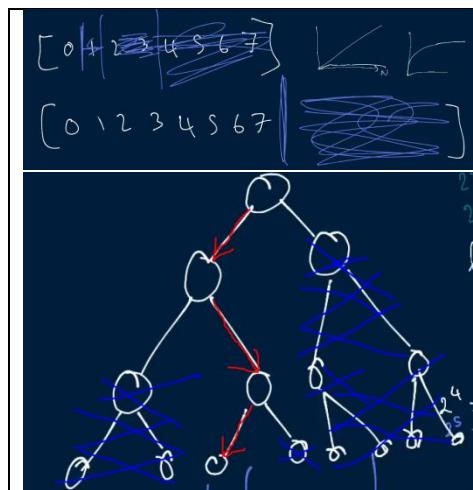
- Notice that the big O notation always refers to the worst case.
- Notice that in case of more than one inputs, for example two input arrays of length n and m, then each variable is significant. For example if the complexity is $O(n^2+m)$ you don't omit the m, since it is a different variable.

$O(\log_2(n))$

In computer science the log refers by default to base 2, not base 10. **As the size of the input increases 2 times, the number of elementary operations increases by a constant unit**. For really large input sizes the number of elementary operations changes very little.

Since $\log(2^0)=0$, $\log(2^1)=1$, $\log(2^2)=2$, $\log(2^3)=3$... $\log(2^n)=n$, if the input array length is 1024 (2^{10}) the complexity is 10, if it is 2048 ($2*1024$ or 2^{11}) the complexity is 11, if it is 4096 ($2*2048$ or 2^{12}) the complexity is 12. Every time the input size doubles, the complexity increases by a constant unit.

Typical $O(\log(n))$ cases



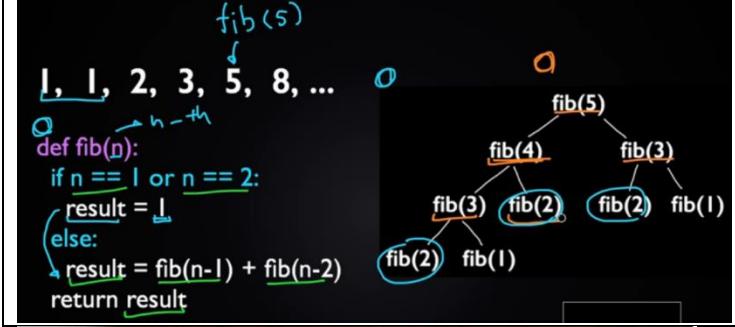
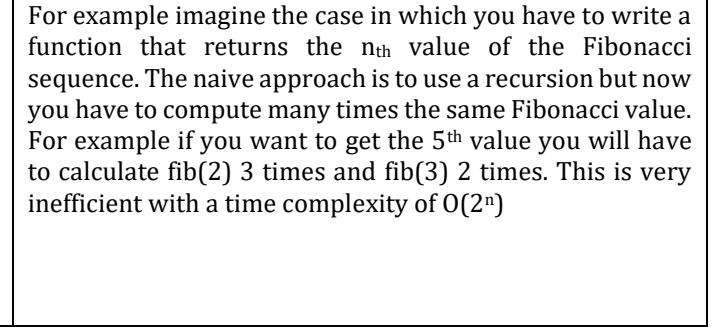
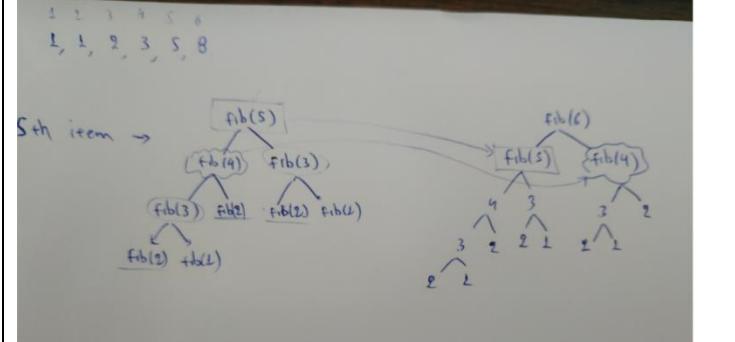
The input is an array of length 8. Your function works in such a way that at every iteration it eliminates half of the array. It would need 3 iterations to eliminate all elements of the array and end up with one value. If the array's length was to double to 16, then you would only need one additional iteration to eliminate one half, and then you end up as before with an array of length 8. So you need 4 iterations in total. Every time the input size doubles you need one additional operation. This is a typical $O(\log(n))$ function.

Another typical example is searching a binary tree. Every time you make a choice, you eliminate half the tree which means you eliminate half the elements. If the tree size was to double in number of elements, this means that the tree would have one additional layer because in a tree roughly half of its elements lie on its last layer. So by adding one more layer you are doubling the size of the tree, which means that you only have to do one additional choice at the end to reach to one element. You are in a $O(\log(n))$ situation.

The way to think in order to decide if you have a $\log(n)$ function is this: Am I eliminating half the input with each elementary operation? If the answer is yes then the function has a time complexity of $O(\log(n))$ assuming I'm not performing more elementary operations after every halving. Or another way to think of it, is if I double the size of the input, I only have to do one additional elementary operation.

$O(2^n)$

In $O(\log n)$ problems when you make a choice you eliminate half the items. In other words, if you double the size of the input you only increase the number of operations by 1. There are problems where the inverse is true. **If you add one more item to the input, you double the size of operations.** These are $O(2^n)$ problems. It's the inverse of a $O(\log n)$ problem. An input of length n needs 2^n operations. An input of $n+1$ needs 2^{n+1} operations which is $2 \cdot 2^n$, or two times the number of operations. An example of such a problem is a Fibonacci calculation function implemented with recursion.

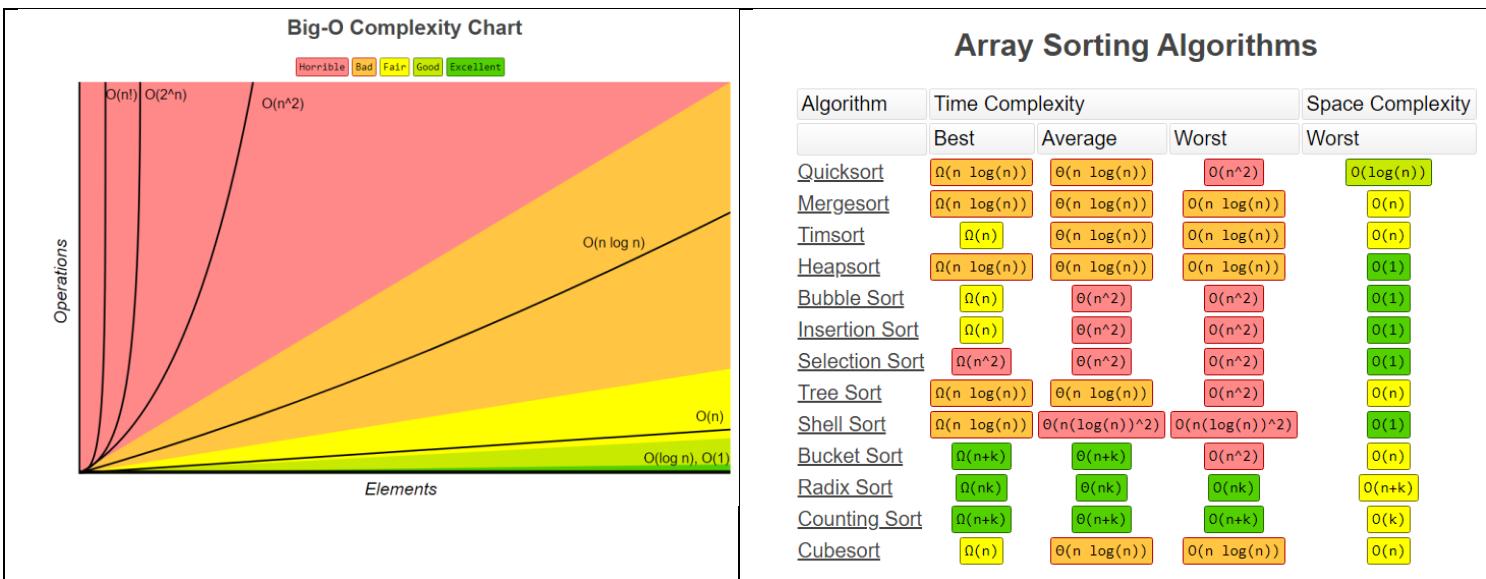
 <pre> fib(5) 1, 1, 2, 3, 5, 8, ... def fib(n): if n == 1 or n == 2: result = 1 else: result = fib(n-1) + fib(n-2) return result </pre>	 <p>For example imagine the case in which you have to write a function that returns the n^{th} value of the Fibonacci sequence. The naive approach is to use a recursion but now you have to compute many times the same Fibonacci value. For example if you want to get the 5th value you will have to calculate $\text{fib}(2)$ 3 times and $\text{fib}(3)$ 2 times. This is very inefficient with a time complexity of $O(2^n)$.</p>
 <p>5th item \rightarrow</p>	<p>Fib(6) is operations of $\text{fib}(5) + \text{all operations of fib}(4)$. It's roughly double the operations of fib(5).</p>

Space complexity

Notice that with space complexity we measure only the space that will be used by the algorithm we measure. For example if a function takes an array of length n and just iterates over it without copying it or creating a new one, its space complexity is $O(1)$ and not $O(n)$. We assume that the array already exists and occupies n memory slots, and our algorithm doesn't use any new memory slots.

Big O notation cheatsheet

<https://www.bigocheatsheet.com/>



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n \log(n))$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	
B-Tree	$\Theta(\log(n))$	$O(n)$								
Red-Black Tree	$\Theta(\log(n))$	$O(n)$								
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	
AVL Tree	$\Theta(\log(n))$	$O(n)$								
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	

Arrays

Arrays

Static and dynamic arrays
Python and javascript don't have static arrays (arrays of predefined known length). for dynamic arrays the OS allocates double the initial length (or something like that)

Initializing an array, which means storing it in memory, is a $O(N)$ for both time and space complexity. For time because you have to write to n memory slots (actually n times the number of bytes per value) so you have n elementary operations and for space since you had 0 memory slots used and then you have n memory slots used (times the bytes per array value).

The orange slots in the picture show how one 64 bits integer is stored. It needs 8 memory slots of 1 byte each.

Inserting to a static array is O(n) for time and O(1) for space complexity. Its $O(n)$ since in order to insert at the start or in the middle or even at the end (if the next slots are full) requires copying the array to another place in memory. In terms of memory the amount of used memory doesn't change ultimately much, since the original array is deleted after the copy.

Dynamic arrays have the same time and space complexity with static arrays except from the appending operation which is O(1). Initially you have some allocated free slots. Appending at that stage is simply $O(1)$. but when they are filled, the OS allocates additional free slots the number of which is equal to the arrays length. This means that it has to copy the whole array to a new place in memory with the appropriate number of slots. Appending when the array is full is $O(N)$ since you have to copy. But then you can append with $O(1)$ until it is full again. This in general is an $O(1)$ operation since the times we have N elementary operations is far less than the times we have one elementary operation. We say that the amortized complexity is $O(1)$.

Pop an element (pop is removing the last element) is a $O(1)$ but removing a random element is $O(n)$ (because the array has to be stored box to box and if you remove an element from the middle you have to copy the array and store it in another place in memory)

But if you want to have an $O(1)$ operation for removing the first item from an array, as an interview tip, you can assume that your array is a Queue from which removing the first item is $O(1)$.

Linked lists

No index (no need to be stored box to box). An element in a singly linked list has a reference of its next element. In a double linked list it has a reference to its previous element too. They have no information about where they are on the list. It's very easy and quick to add or remove elements in comparison to arrays that have an index and maybe even a fixed length.

The big difference of a linked list with an array is that there is no need for the elements of the linked list to be stored box to box. Each element is stored anywhere in memory, but we need to use one additional memory slot for each element which would be the memory address of the next element. So each element needs 2 box to box memory slots. The last element of the linked list has a pointer that points to the null memory address. The first element is called the head of the list.

Accessing an element is $O(i)$ in time complexity, where i is the "index". To access an element we have to begin from the head of the list and follow the pointers until we find the i^{th} element.

Inserting and deleting though are $O(1)$. This is the main advantage of a linked list. Inserting a new head is simply adding two new memory slots, one for the new element and one for the pointer to the previous head. Inserting in the middle requires three operations, two new memory slots for the value and the pointer to the next value and the change of the previous pointer to the new value. But you have to traverse the list in order to find the required "index" so it is an $O(i)$ operation. If you have a reference to the tail of the list, adding a new tail is a $O(1)$ operation otherwise it's $O(N)$ since you have to traverse the whole list.

The pointer of a singly list is called the *next* pointer. Next and *prev* in the doubly list.

Usually languages don't have a build in implementation of linked lists, which means that you have to create them or use a 3rd party library.

Stacks and Queues

For both we have: Insert O(1) ST. Search O(1) T, O(n) S

They can be implemented with linked lists or dynamic arrays. Each has its own pros and cons.

Stack:

- Array: As a dynamic array
- Linked list: As a singly-linked list with a head pointer.

Queue:

- Linked list: As a singly-linked list with a head and tail pointer.
- Array: As a circular buffer backed by an array.

Stacks and queues

Queues can be implemented with linked lists, with reference to head and tail in order to have enqueue and dequeue in constant time.
Add and remove from queue is typically called enqueue and dequeue. They also have a peek method. They are O(1) operations.

Stacks can be implemented with dynamic arrays. Add and remove from stack is typically called push and pop. They also have a peek method which returns the element to be popped without actually deleting it. They are O(1) operations.

Although they are pretty simple data structures you can make them more sophisticated if you want. For example you can have Max/Min stacks, where you keep a reference to the max or min element of the stack.

Stacks

Imagine a stack of books in a table you add one on the top and you remove from the top too. The stack is a LIFO data structure where the last item that was inserted in the stack is the first that will be extracted from the stack. There are no other requirements for the elements of a stack.

Implementation using python lists

Python's built-in data structure list can be used as a stack. `append()` is used to add elements to the top of the stack while `pop()` removes the element in LIFO order. Unfortunately, the list has a few shortcomings. The biggest issue is that it can run into speed issues as it grows. The items in the list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some `append()` calls taking much longer than other ones.

Implementation using `collections.deque`

Faster insert and delete O(1) instead of O(N) in lists

Queues

FIFO structure. The opposite of the stack. It has a Head and a Tail elements. Add to the tail is called enqueue. Removing from the head is called dequeue. Another operation is peek where you just look at the head element without removing it.
A double ended queue is called deque.

A Priority queue. Each element has a priority property too and when you remove an element you remove the one with the highest priority. It is very efficient to do this.

Implementation using python list

list.append() and list.pop(0)

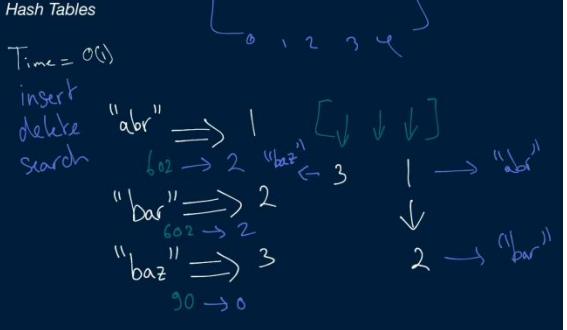
Hash tables

These data structures are usually embedded in languages. In javascript its the javascript object while in Python is the dictionary. The following is a description of how they work under the hood. The important thing is that **insert**, **delete** and **search** are all constant time operations **O(1)**.

Under the hood, a hash table uses a dynamic array of linked lists to efficiently store key/value pairs. When inserting a key/value pair, a hash function first maps the key, which is typically a string (or any data that can be hashed, depending on the implementation of the hash table), to an integer value and, by extension, to an index in the underlying dynamic array. Then, the value associated with the key is added to the linked list stored at that index in the dynamic array, and a reference to the key is also stored with the value.

You have three key value pairs that you want to store in a way that you can access them easily by key. To do so, you have to store the values in an array of length 2, since arrays have constant time complexity for element retrieval. To do so you have to find a way to convert the keys to indexes. This is the job of a function called "hash function".

A typical hash function: 1. String to int 2. int % (array length)	A hash function gets a value as an input and encodes it to an integer that can be used as the index of an array. In the general case the value can be anything, for example a string.
---	---

	In this case the string is converted to an integer, for example by getting the ascii encoding of each character and adding them together. This would result to a number for example 602. But in the previous example you want to store the values in an underlying array of length 2. So one way to convert an integer to a number between 0 and m-1, is to get the modulo of the integer with the value m . In this case we take the modulo of 602 with the length of the array which is 3, to get a value between 0 and 2. $602 \% 3 = 2$. So the value of this key, will be stored in index 2 of the underlying array. (<u>Apart from the value, you can store a pointer to the memory address that contains its key</u>). So this way you can retrieve a value by a given key in constant time (access array is constant time).
---	---

Notice that you might have index collisions where different keys are encoded to the same integer. In these cases the value at that index would be a linked list and retrieving from it its $O(\text{len of list})$ since you have to traverse the list.

When the underlying array is full with linked lists, you can implement the resizing of the array. All of its elements are copied, a new array with double the size is stored, and all elements pass from a new hash function with the new array length, giving new indexes more spread out. You can have resizing as you delete elements from the array too. Have in mind the load factor of a hash table's underlying array: Load Factor = Number of Entries / Number of Buckets. A value of the array (which could be a list of values) is called a bucket in this context. The closer to 0 the emptier the hash table so you probably would like to resize to reduce the number of buckets. Closer to 1 you need to resize to increase the buckets.

In the worst case scenario all values would collide to the same index and you would have access time of $O(N)$. This is the worst case. But in general you can assume that the hash functions used and the hash table implementation are very sophisticated so that you rarely have collisions. So you can assume that you always have the average case which is access with $O(1)$.

A map is a collection of key value pairs, so a hash table that stores key value pairs can also be called a hash map.

About hash functions

There are a lot of hash functions that can do this. Which one to select depends on the case. For example if you want to implement a word dictionary (where you have a word as key and the definition as the value) then you need unique hash values meaning that you don't allow collisions. Such a hash function is this one:

$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$ $\text{"HASH"} \rightarrow 72 * 31^3 + 65 * 31^2 + 83 * 31 + 72 =$ $(72, 65, 83, 72)$	You have the ASCII value for each letter. You multiply each ascii value with a power of 31. This operation produces unique hash values for any string. (31 is a number that has this property. There are also other numbers like this too) But the downside is that the hash values are very large so you need a lot of space. Even for 4 letter strings the hash values are already quite big.
---	---

Strings

Strings are actually a data structure. You can do a lot of operations on them and doing operations has time and space implications.

Strings are stored in memory as arrays of characters. A character is encoded to an integer and stored. There are various encoding standards. A common one is the ASCII encoding standard, which encodes to less than 256 integers. This means that each character can be stored in one byte (8 bits) since one byte can have 256 (2^8) combinations. If you need to use more than 256 characters, then you need another encoding that will use more than one byte per character.

Access a character is like accessing an index of the array so it is O(1) ST, Traverse is O(n)

Inserting depends on the language implementation of strings. It is different for mutable and immutable strings.

In some languages like C++ strings are mutable. In Python and Javascript they are immutable. This means that when you modify a declared string, you actually copy the existing string. This is a O(N) operation. Inserting to a string like `str += "_1"` is a O(N) ST, since the string must be copied. Adding two strings is O(n+m) ST.

These languages offer workarounds to have O(1) operations instead. To do so, you split the string to an array of characters. Now you don't have a string but an array. Append and delete on an (dynamic) array is O(1). You do the operations on the array and re concatenate back to a new string at the end.

```
My_list = list(string)
```

```
My_string = ".join(my_list)
```

Graphs

Complexity analysis

Store: O(V+E) analogous to the number of edges plus the number of vertices.

Traversing: O(V+E) T,S both for DFS and BFS

DFS: seen list + a stack

BFS: seen list + a queue

It is a data structure designed to show relationships between objects. It is a collection of nodes that may or may not be connected to one another. Nodes are called vertices and the connections edges.

They can have no root node since graphs can have cycles (you start at a node and follow edges back to the same node). Edges can contain data too, for example they can show the strength of the connections (in case of people, how many times they have met for example). A Tree is a special type of graph

Connectivity, Direction and Cycles are three important properties of a graph.

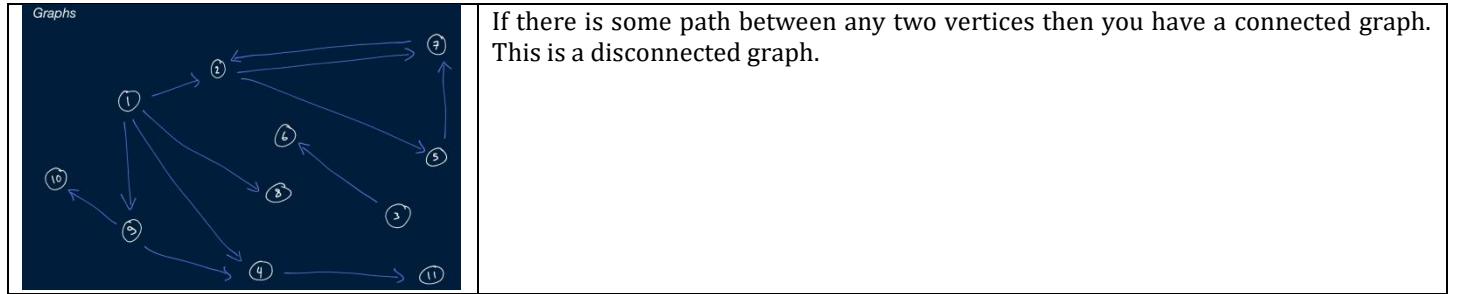
➤ Directed Graphs

Edges can have direction meaning that the relation applies only in one direction. Graphs with such edges are called directed graphs.

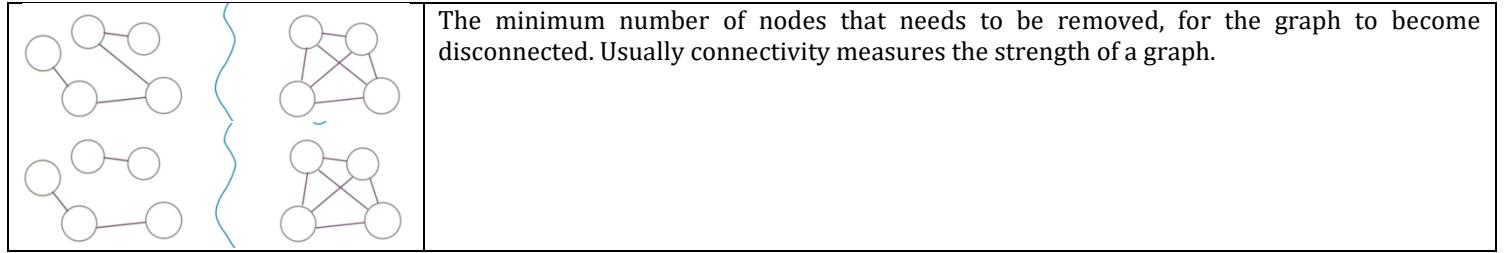
➤ Acyclic graphs

Graphs with no cycles. If you traverse a graph following edges, and you end up in a vertex that you have already visited then this graph contains a cycle (and you have to skip the already visited vertex in order to continue traversing the graph)

➤ Connected graphs



Connectivity



Strongly and weakly connected

A digraph is strongly connected if every vertex is reachable from every other following the directions of the arcs. I.e., for every pair of distinct vertices u and v there exists a directed path from u to v .

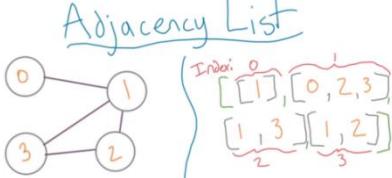
A digraph is weakly connected if when considering it as an undirected graph it is connected. I.e., for every pair of distinct vertices u and v there exists an undirected path (potentially running opposite the direction on an edge) from u to v .

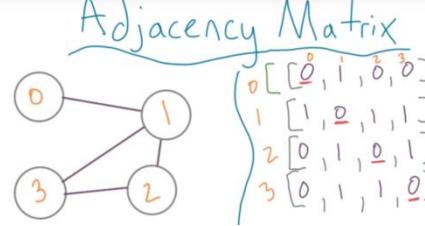
Node degree

the number of edges connected to a particular node

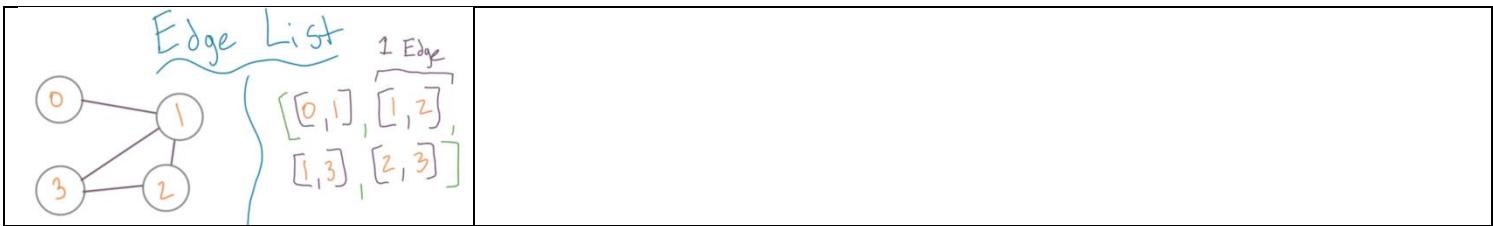
Code representations of graphs

This is a very common implementation. Every node stores a list of its edges, or in other words, of its adjacency, hence the term adjacency list.

 <pre>{ "zero": ["one"], "one": ["zero", "two", "three"], "two": ["one", "three"], "three": ["one", "two"] }</pre>	<p>First implementation type</p> <p>One type of implementation of storing a vertex would be to store its value and a dynamic array (or a pointer to a dynamic array) where the array isn't a list of pointers of adjacent vertices but a list of the actual values of the adjacent vertices.</p> <p>if the values of the vertices can be used as the index of an array, you can implement a graph with an <u>array of arrays</u>. An alternative would be with a hash table where the keys are the values and the values are the list of adjacent keys.</p> <p>Array with arrays: The value of the vertices (which in these examples are integers starting from 0 so they can be used as index of a list) correspond to the index of a list. And the value at a specific index is another inner lists that contains the ids of the vertex's adjacent vertices. An adjacent vertex is one that is connected with vertex in question by an edge.</p> <p>Hash table: An alternative implementation is using a hash table (for example a python dictionary) where the key is the value of the vertex and the hash table's value is the adjacency list (a list of adjacent keys).</p>
<pre>zero, one, two, three = {}, {}, {}, {} zero = {"zero": [one]} one = {"one": [zero, two, three]} two = {"two": [one, three]} three = {"three": [one, two]}</pre>	<p>Second implementation type</p> <p>An other type of implementation of storing a vertex, would be to store its value and a dynamic array (or a pointer to a dynamic array) but now the dynamic array contains pointers to the adjacent vertices (the memory addresses of the adjacent vertices).</p> <p>Python doesn't support pointers, which means that you can't externally define variables that point to the specific memory location of another variable. But you can do it indirectly.</p>
<pre>dict1 = {'first': 'hello', 'second': 'world'} dict2 = dict1 # pointer assignation mechanism dict2['first'] = 'bye' dict1 >>> {'first': 'bye', 'second': 'world'}</pre>	<p>Javascript and Python don't externally use pointers. But you can use them indirectly using javascript objects or python dictionaries.</p> <pre>//this will make object1 point to the memory location that object2 is pointing at object1 = object2;</pre> <p>Unlike in C, you can't see the actual address of the pointer nor the actual value of the pointer, you can only dereference it (get the value at the address it points to.)</p>

	<p>Adjacency Matrix</p> <p>Essentially, vertex ids correspond to the index of both the outer and the inner list. When there is an edge between two nodes the number 1 is used.</p> <p>There would be 1 in the diagonal if there was edges that start and end in the same node. (Are this kind of edges valid?)</p> <p>A single edge show up twice in this matrix.</p>
---	--

You can use objects for vertices and edges to represent the graph but this set up makes traversing a graph less efficient since you have to search through many objects. Another way to represent a graph is to represent the edges information in a list.



Depending on the most often use case you choose the respective representation. If you regularly search for node degrees (the number of edges connected to a particular node) then an adjacency list would be the most convenient way to represent a graph

Traversing Graphs

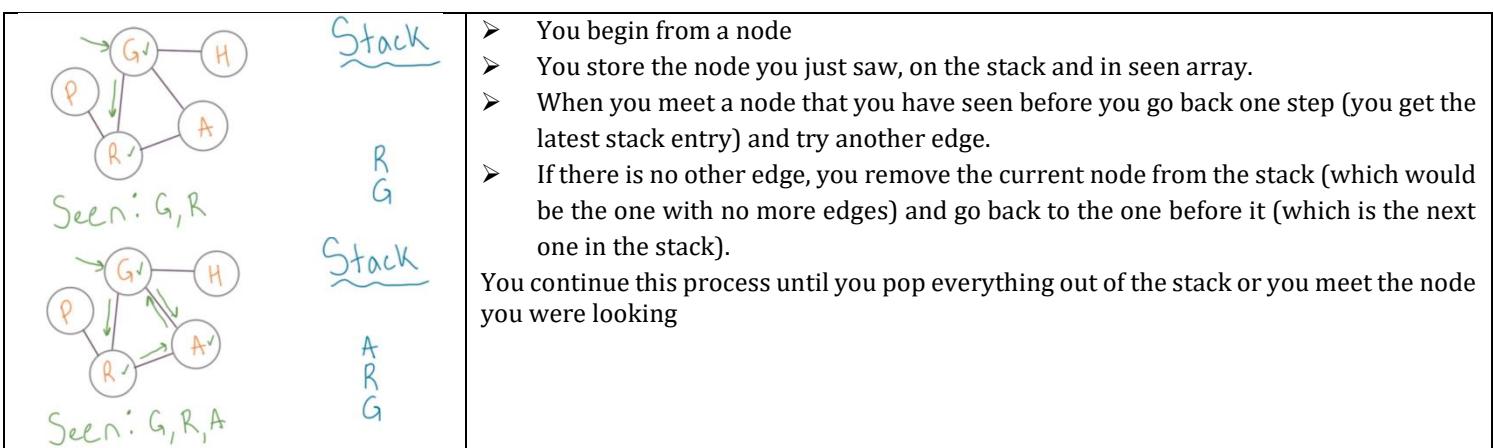
In a traversal operation you go through all the elements of a data structure while on search you stop when you find what you were looking for. Very important issue to know how to write DFS and BFS searches.

Graphs are very convenient for storing relationships and also because it is easy to traverse them based on connections. There are 2 types of traversal algorithms. Depth First Search DFS (you search all the way down a path) and breadth first search BFS (you search all adjacent vertices before moving deeper, you search every edge of a node before continuing to the next node). Since there is no root, you choose a random node to start traversing.

➤ DFS

You can implement a DFS algorithm using a “seen” list for nodes and a **stack**. Another approach is to replace the stack with recursion.

The efficiency is $O(E+V)$ it is analogous to the number of edges plus the number of vertices.



➤ BFS

First you visit all children of the selected node and then pick the first child and visit all of its children, then the children of the second child etc.

You can implement a BFS algorithm using a **seen** list for nodes and a **queue**. When a node is marked as seen is added to the queue. $O([E]+[V])$ the efficiency analogous to the number of edges plus the number of vertices.

<p>Seen: G, H</p> <p>Queue H</p> <p>Seen: G, H, A</p> <p>Queue H A</p>	<p>You start from a random node you mark it as seen and you move to the next one. You mark it as seen and you add it to the queue. When you mark a node as seen you add it to the queue. When you run out of edges you dequeue a node from the queue and use that as the next starting place.</p> <p>Queue</p> <p>H</p> <p>Seen: G, H, A, R</p>
<p>Seen: G, H, A, R</p>	<p>A BFS algorithm is equivalent of transforming a graph to a tree where the root is the node from which BFS started.</p>

Standard Graph Paths

➤ Eulerian path

A path that traverses every edge of a graph exactly one time (allowing for revisiting vertices).

A graph can have an Eulerian path if the only two vertices with odd degree (number of edges) is the starting and ending vertex.

➤ Eulerian cycle

It is an Eulerian path that starts and ends on the same vertex.

A graph can have an Eulerian cycle only if all vertices have an even degree (an even number of edges)

<p>ABC</p> <p>ABCDE</p>	<p>Not every graph is capable of having an Eulerian path.</p>
	<p>There are algorithms for finding Eulerian paths and cycles in a graph.</p> <p>This algorithm we combine two paths. $O(E)$</p>

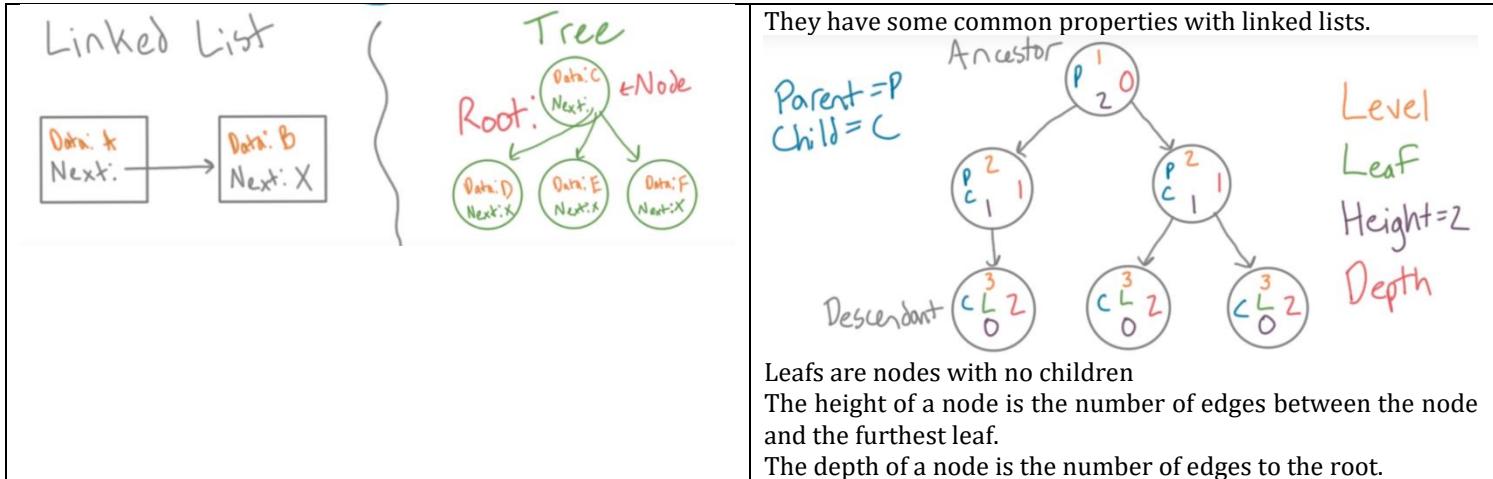
➤ Hamiltonian Paths

A path that must visit every vertex exactly one time. And Hamiltonian cycle will start and end in the same vertex. Trying to identify if a graph has a Hamiltonian path is a famous computer science problem.

Trees

It is a rooted, directed, acyclic, connected graph where each node can only have one parent.

- The **height** of a balanced tree with n nodes is of length $\log(n)$
- The **last layer** of a full tree contains approximately **half the trees nodes**
- In binary trees, the parent with index n has children in indexes $2n+1$ and $2n+2$
- You want balanced trees so that the **search time is $O(\log n)$**



- They must be fully connected which means that if you are starting from the root there must be a path for every node of the tree. Each path must be unique?
- There must be no cycles in a tree. There must be no way to encounter the same node twice for example starting from the root and ending back to the root.
- The nodes have no specific order, each tree can have different ordering rules depending on the algorithm used

Branch: any path that starts from the root node and goes to a leaf is a branch

Complete tree: a tree which is filled in all levels except from the last level which can be not filled , but must be populated left to right.

Full tree: a k-ary (binary, tertiary etc.) tree is full if all nodes have k children or no children at all

Perfect tree: a tree where all leaf nodes have the same depth.

Tree traversal

Trees have no particular order so there are two different approaches when you want to traverse a tree.

- Depth first search (DFS): if there are children nodes to a node, exploring them is a priority
In order search, pre order, post order
- Breadth first search (BFS): the priority is visiting the nodes of the same level before we visit childnodes.

Complexity

Store: $O(N)$ S

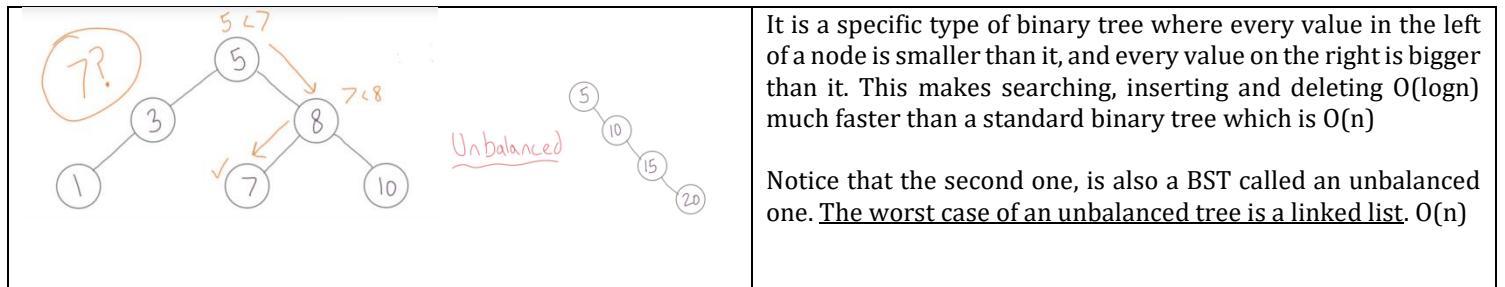
Traverse: $O(N)$ T

Search: **O(n)** for binary trees, **O(log(N))** for Binary Search Trees

Binary trees

A node can have at most two children. Search is $O(n)$. delete is $O(n)$. Insert is faster

Binary Search Tree



Properties of a BST

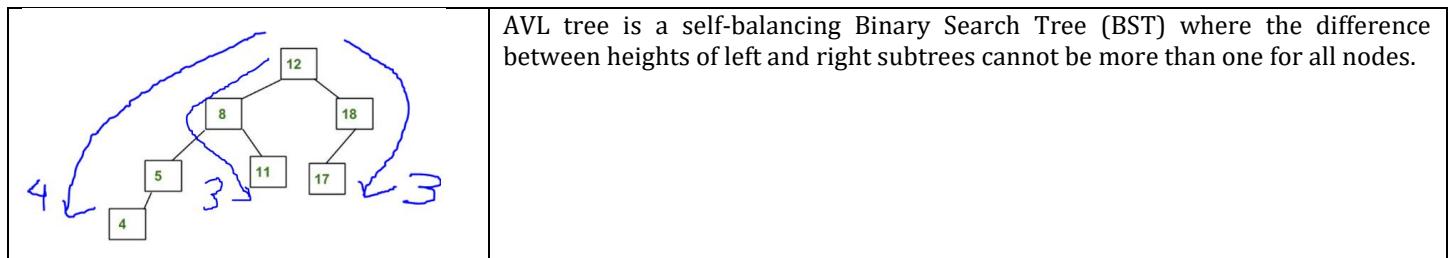
- The left subtree of a node has nodes which are only lesser than that node's key.
- The right subtree of a node has nodes which are only greater than that node's key.
- Each node has distinct keys and duplicates are not allowed in Binary Search Tree.
- The left and right subtree must also be a binary search tree.

If you want to check where an element lies you compare its value with the current node. If it is larger then it goes to the right. If it is smaller then it goes to the left. If it is the same then you have found it.

In binary search trees, at every step you make a choice and eliminate one subtree of the tree. If the tree is balanced then you are eliminating half the nodes at each step. But if the tree is unbalanced, very skewed to one side then you wouldn't eliminate half the nodes at each choice so the complexity would be worst. Having a **$O(\log n)$ time complexity for search, is a very important property for a tree** so there are some sophisticated tree implementations like the Red black trees or AVL trees that rebalance themselves at every operation (addition or deletion for example) so that they keep exhibiting $O(\log(n))$ search complexity.

Important: Most of the BST (binary search tree) operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that the height of the tree remains $O(\log n)$ (balanced tree), after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. This is what AVL trees and Red-black trees achieve. They are self-rebalancing trees. After every insertion or deletion there are certain operations that take place to rebalance the tree (certain "rotations" around specific nodes of the tree)

AVL tree



Red Black tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black).

<p>Following are NOT possible 3-noded Red-Black Trees</p> <p>Violates Property 4</p> <p>Violates Property 4</p> <p>Violates Property 3</p> <p>All Possible Structure of a 3-noded Red-Black Tree</p> <p>Proper structure of three noded Red-black tree</p>	<p>Following are possible Red-Black Trees with 3 nodes</p>	<p>Rules That Every Red-Black Tree Follows:</p> <ol style="list-style-type: none"> Every node has a color either red or black. The root of the tree is always black. There are no two adjacent red nodes (A red node cannot have a red parent or red child). Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes. All leaf nodes are black nodes.
--	---	---

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over the Red-Black Tree.

Heaps

A heap is a specialized **tree-based data structure** that satisfies the **heap property**: If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. Or in other words:

<p>Example of a complete binary max heap</p>	<p>Example of a complete binary min heap</p>	<p>Each parent element's key must be \geq or \leq to its children. In the first case it's a max heap and in the second a min heap.</p> <p>A binary heap it's a heap in which a parent can have only 2 children.</p> <p>Min-heaps are often used to implement priority queues. The children elements of a parent are called siblings. The elements on the same level are called cousins. Notice that the ordering of siblings and cousins of a level is not specified by the heap property.</p>
--	--	---

There are some common operations that can be applied to a heap: **find-max/min**, **merge heaps**, **insert** a key. Its advantages for priority queues are obvious. You can have priority organized in parent-children level. This can't be done with a simple array.

Inserting to a heap is $O(\log n)$.

Heapify

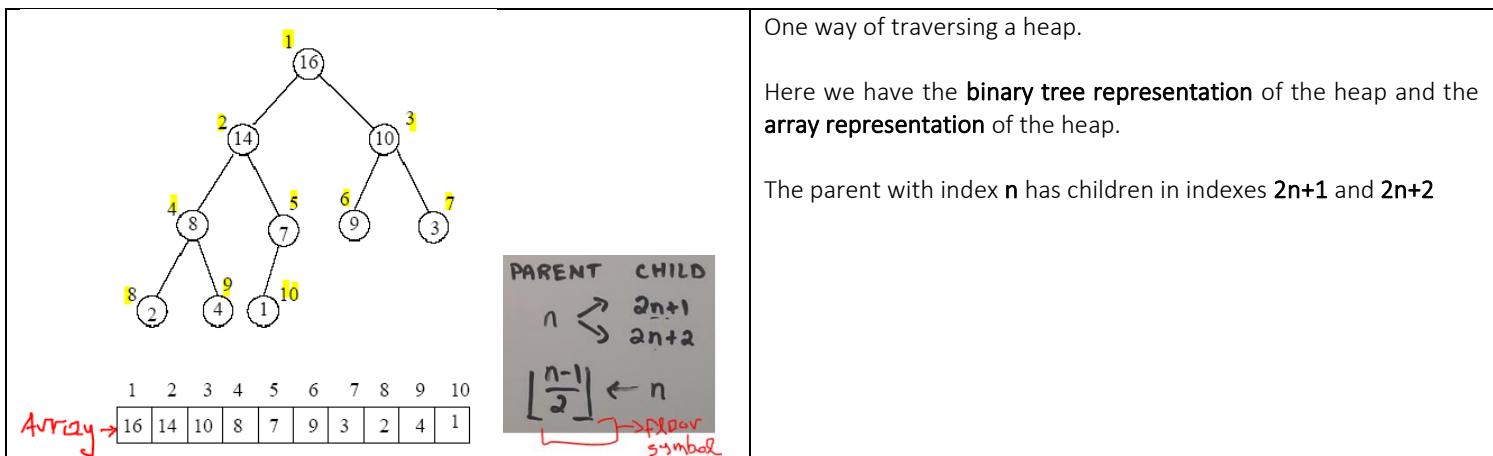
Reorder the heap according to the heap property (max or min values). After an insertion of a node heapify is required.

Extract

The operation of removing the head node and reordering the heap.

Heaps are often stored as arrays because its more space efficient than storing as a heap

<u>Tree</u> 	<u>Array</u> 	<p>When its stored as an array you only have to store the value and that along with its index completely describes the heap (since you have a convention of filling a level for example from left to right). In order to store it as a heap you need more properties for each element of the heap. The array avoids storing the pointers (left, right, parent)</p>
-----------------	------------------	--



Tries

A type of tree

Balanced and unbalanced trees

In balanced trees the nodes are condensed in a few levels while unbalanced nodes are spread out on many levels. In general you would prefer balanced trees since they are more efficient in terms of search, insert and delete. For this reason there are trees with certain rules that make the tree to be a self-balancing one. Meaning insertion and deletion of nodes are made in such a way that the tree remains balanced. One type of such trees is the red-black trees.

Red Black trees

They are also BSTs so they follow the rules of both the red-black and BSD trees.

Tips

Dynamic arrays: insert $O(N)$, append $O(1)$, access $O(1)$, search $O(1)$,

Linked lists: insert $O(1)$, access $O(N)$,

Hash tables: **insert**, **delete** and **search** are all constant time operations **$O(1)$** .

Stacks and Queues: Insert $O(1)$, search $O(1)$

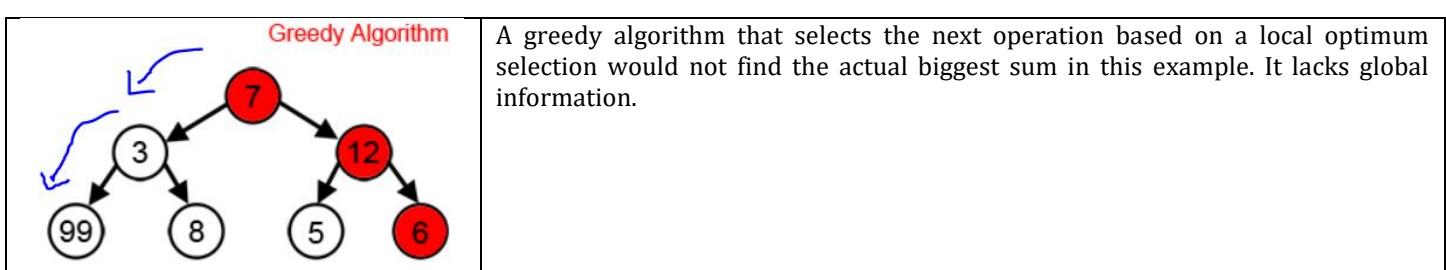
one way to convert an integer to a number between 0 and $m-1$, is to get the modulo of the integer with the value m . In this case we take the modulo of 602 with the length of the array which is 3, to get a value between 0 and 2.

Algorithms

Algorithms for searching and sorting on different data structures. Brute-force greedy algorithms, graph algorithms, dynamic programming

Greedy algorithms

Greedy algorithms make locally optimal choices at each stage with the hope of finding the global optimum



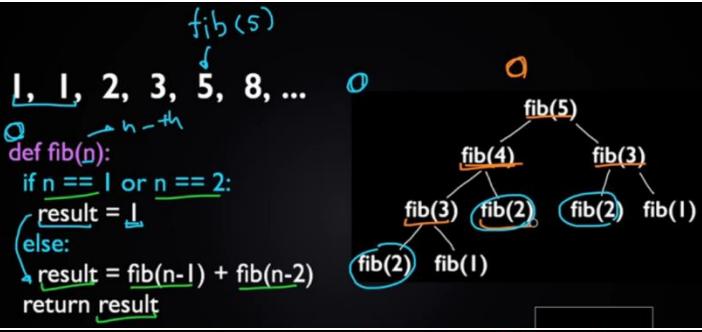
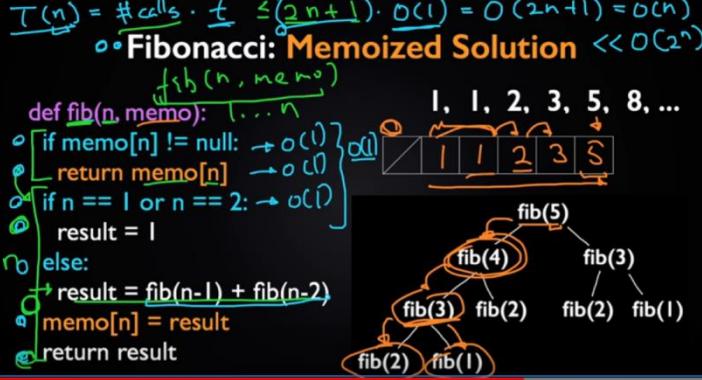
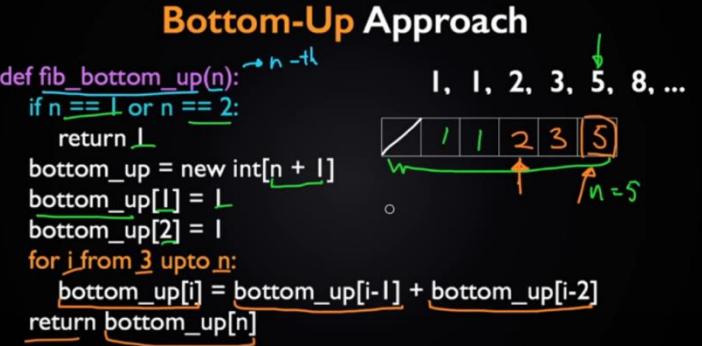
NP-Hard problems

They are problems to which there is no known solution that can solve them in polynomial time $O(n^2)$, $O(n)$, $O(2)$. for these problems there are two approaches, exact and approximation algorithms. In the first class of solutions you can find the exact right answer but in no polynomial time while in the second class you can find an approximation of the optimal solution much faster and in some cases in polynomial time.

Dynamic programming (Memoization in recursion)

It is a way of doing your algorithm more efficient by storing some of the intermediate results. Find a recursive solution for your problem and try to find a lot of repeated states in order to store them so that they are not calculated at every step.

It works well when your algorithm has a lot of repeated computations so using dynamic programming approach you don't have to repeat them each time.

 <pre> fib(5) ↓ 1, 1, 2, 3, 5, 8, ... ↓ def fib(n): if n == 1 or n == 2: result = 1 else: result = fib(n-1) + fib(n-2) return result </pre>	<p>For example imagine the case in which you have to write a function that returns the nth value of the Fibonacci sequence. The naive approach is to use a recursion but now you have to compute many times the same Fibonacci value. For example if you want to get the 5th value you will have to calculate fib(2) 3 times and fib(3) 2 times. This is very inefficient with a time complexity of $O(2^n)$</p>
 <pre> T(n) = #calls · t ≤ (n+1) · O(1) = O(2n+1) = O(n) o Fibonacci: Memoized Solution << O(2^n) def fib(n, memo): if memo[n] != null: → O(1) return memo[n] → O(1) if n == 1 or n == 2: → O(1) result = 1 else: result = fib(n-1) + fib(n-2) memo[n] = result return result </pre>	<p>Another approach is to use a list to store the Fibonacci values when you calculate them and read the stored value every time you need it. This approach is called Memoization and it is $O(n)$ which is a big improvement. Notice though that you might reach recursion limits for the programming language that you use for very large value of n.</p>
 <pre> Bottom-Up Approach def fib_bottom_up(n): if n == 1 or n == 2: return 1 bottom_up = new int[n + 1] bottom_up[1] = 1 bottom_up[2] = 1 for i from 3 upto n: bottom_up[i] = bottom_up[i-1] + bottom_up[i-2] return bottom_up[n] </pre>	<p>Another approach is called bottom up approach. You basically create the list from scratch every time and just return the nth element. You don't use recursion. $O(n)$</p>

Searching and sorting

- Searching
- Binary search

You must have a sorted array. You take the middle element and compare it with the element you want to find. You repeat the process in one half of the array.

It has an efficiency of $T=O(\log n)$

Linear search

It is a Sequential Search algorithm: The list or array is traversed sequentially and every element is checked. $T=O(n)$

Recursion

<pre> function recursive(input): if input <= 0 return input else output = recursive(input - 1) return output </pre> <p><i>② Base Case</i></p> <p><i>①</i> <i>③</i></p>	<p><i>Our Steps</i></p> <pre> recursive(2) recursive(1) recursive(0) return 0 output = 0 return 0 output = 0 return 0 </pre>	<p>A function calling itself. You need a base case which is the exit condition. You also need to call the function with a different input from the one it was called otherwise you will have an infinite recursion.</p>
--	--	---

Notice that an upper function call would not return any value until its lower function call returns.

➤ Sorting

Array Sorting Algorithms				
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

There are in place sorting algorithms where you don't have to copy the elements to a new sequence during the sorting process. You just work on the given sequence. This approach offers better space complexity (you don't use much memory) but worst time complexity (you need more processing).

(Merge sort is a fair choice)

Bubble Sort

In place algorithm. You begin from the first element and compare it with the second. If is bigger you move it to the second place. Then you compare it with the third and so on. Its a naive approach. You compare each element with all the others.

	<p>For n elements you must do $n-1$ iterations $n-1$ times so its efficiency (its time complexity) is $T=O(n^2)$. We didn't use any other sequence so the space complexity is $S=O(1)$. No matter how big n is, the space needed would not be increased by the algorithm.</p>
--	---

Merge Sort

Not an in place algorithm

time complexity is **T=O(nlogn)** but the space complexity is **S=O(n)**. You make n comparisons log times. Log times since you for double the size, you only need to split one more time. You use new n new memory slots for the new arrays so space complexity id O(n).

	<p>You split the sequence to sequences of length one, you then merge them by two and sort the two in each sequence. Then you merge in four by comparing the first elements of the sequences and continue merging until the end.</p> <p>How you sort two sequences that are merged in one? You compare the first elements. You put the smallest item in the last position of the merged array and remove it from its original array. Then you compare again the first elements and repeat. The smallest of the two will be put to the last position of the new array. If one array becomes empty, you append what remains of the other array in the last position of the new array. The more elements that remain in the end in one array the better for the time efficient of the algorithm.</p>
--	--

Quick sort

In place algorithm

Worst case time complexity is **Tworst=O(n²)** so it is not good for near sorted sequences but average time complexity is **Tavg=O(nlogn)**

	<p>You select a random element (typically the last) and move all the bigger elements to the right and smaller to the left. The selected element is called <u>the pivot</u>. Then you do the same thing in the two sequence segments around the first pivot by selecting a new pivot for each and continue until the sequence is completely sorted.</p>
--	--

There are more sorting algorithms

Shortest path

Finding the shortest path in a Graph. The shortest path between two nodes is the one where the sum of the weighted edges is the smallest of all others (you might have weights in the edges).

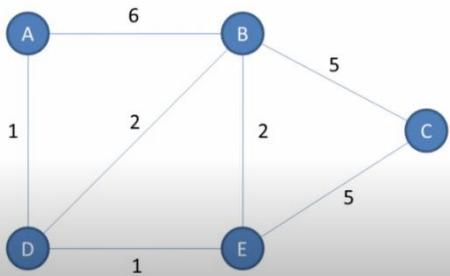
For unweighted edges finding the shortest path requires running a BFS on the graph.

Dijkstra's Algorithm

The objective f the algorithm is to find the shortest path between any two vertices in the graph. In particular it will find the shortest path between a starting vertex and all other vertices.

T=O(|V|^2). One solution to finding the shortest path of a weighted undirected graphs. We give all vertices a **distance value**. A distance value is the sum of all weights in a path between the starting point and whatever vertex we are on. We also use a **minimum priority list** where the priority value is the distance value. At a particular step we pick whatever looks best at the moment.

Find the shortest path from vertex A to every other vertex



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

The solution could be written in a table with three columns

Shortest distance

This column shows the shortest distance (not the actual path that gave that distance) of the vertex from the starting vertex

Previous vertex

The previous vertex column gives us the actual (shortest) path between the starting vertex and all other vertices.

For example lets examine C. its shortest distance from A (the starting point) is 7. let's find out that path. The previous vertex of C is E. let's go to E. It's previous is D. it's previous is A. so the path is A->D->E->->C

The algorithm

We have to fill in that table. We start from A. We keep two lists, visited and unvisited vertices.

Vertex	Shortest distance from A	Previous vertex
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

We begin from A. its shortest distance to A is 0. the shortest distance for the others is unknown so we assume its infinite. This is the starting point of the algorithm. We now select the vertex from the unvisited that has the smallest distance from the starting vertex.

Let distance of start vertex from start vertex = 0
Let distance of all other vertices from start = ∞ (infinity)

Repeat

Visit the unvisited vertex with the smallest known distance from the start vertex

For the current vertex, examine its unvisited neighbours

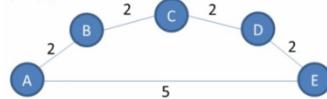
For the current vertex, calculate distance of each neighbour from start vertex

If the calculated distance of a vertex is less than the known distance, update the shortest distance

Update the previous vertex for each of the updated distances

Add the current vertex to the list of visited vertices

Until all vertices visited



It is a greedy algorithm due to the first step. It selects the vertex with the smallest distance. This is a locally optimum decision. Although, the algorithm explores all nodes so it will ultimately find the shortest path no matter what. Me: if we search for the smallest path between two particular nodes, lets say, A and E we could modify the algorithm and stop it as it moves along the curve (a->b->c etc.) if the distance becomes bigger than 5.

Let distance of start vertex from start vertex = 0
 Let distance of all other vertices from start = ∞ (infinity)

WHILE vertices remain unvisited

 Visit unvisited vertex with smallest known distance from start vertex (call this 'current vertex')

 FOR each unvisited neighbour of the current vertex

 Calculate the distance from start vertex

 If the calculated distance of this vertex is less than the known distance

 Update shortest distance to this vertex

 Update the previous vertex with the current vertex

 end if

 NEXT unvisited neighbour

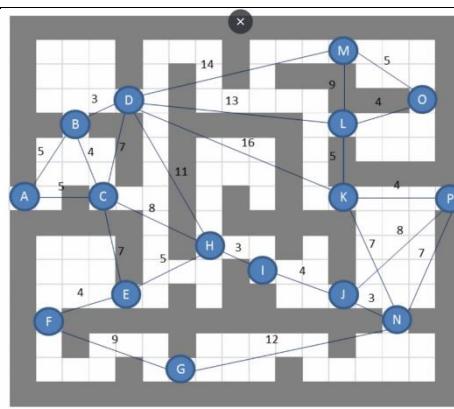
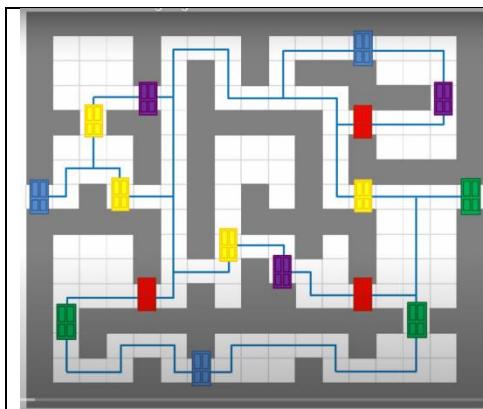
 Add the current vertex o the list of visited vertices

END WHILE

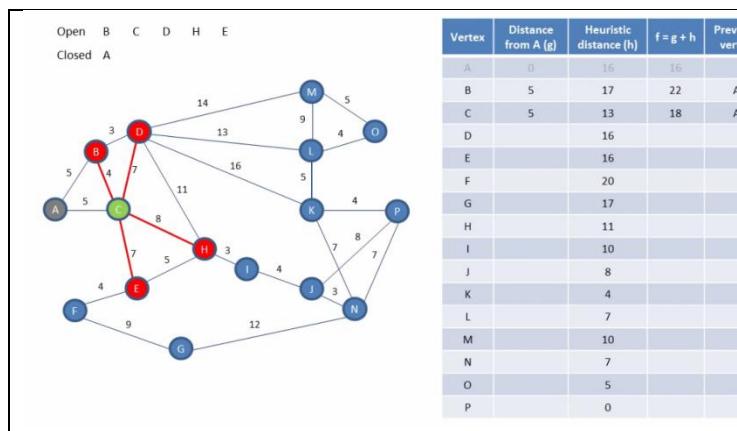
The algorithm in some more detail

A* pathfinding algorithm

An algorithm for finding the shortest path between two particular nodes in a graph. It is an enhancement of the dijkstra's algorithm potentially more efficient for finding the shortest distance between two particular nodes. Dijkstra's algorithm finds the shortest distances between all nodes of the graph and this might be more information that we need in some cases. A* doesn't have to visit every node if the graph. A* will find a path if a path exists, but how efficiently will do it, depends on chance (since you randomly select nodes of equal distance) and the quality of the heuristics.



Notice how we can turn a problem to a graph problem. In a 2d game, where we have doors or checkpoints, we can symbolize them as graph vertices where the distance between them is the number of steps. So if we want to find the shortest path between our position and a specific checkpoint, we can implement A* algorithm on the graph.



Summary

- A* has a wide range of applications
- A* finds the shortest path between two vertices
- A* does not have to visit all vertices, ideally
- A* picks the most promising looking node next
- The better the heuristic, the quicker A* finds the path
- Heuristic is problem specific
- Open nodes known as 'the fringe' or 'the frontier'
- List of open nodes can be implemented as a priority queue
- Each node on the path keeps track of the one that came before
- A* will always find a solution if one exists

Heuristics

A* algorithm contains a Heuristic estimate element (an educated guess) which is important for its efficiency. It is an estimate of the remaining distance from the current node to the target node. This heuristic distance can be calculated upfront with some context that we have about the specific problem. For example in te case of the 2d game we can calculate the euclidian distance

between the checkpoints. Notice that if there are blockages (bombs for example) in the path, or portals that transport you in another position then the Euclidean distance might not be a good heuristic value.

- If the heuristics are 0 then all nodes will be visited (which is what Dijkstra's algorithm does).
- If they are too large, A* will find a path if it exists, but it might not be the shortest path.

The Travelling Salesman Problem

What is the fastest path to visit all nodes of a graph and return to the starting node. It is one of the most famous problems in computer science. It is an optimization problem. It is an NP-hard problem. Some exact solution algorithms:

- Brute force approach
- Held-Karp Algorithm (dynamic programming)

Some approximation algorithms:

- Christofides algorithm

The path found would be at most 50% longer than the fastest route.

Misc

Knapsack (σακίδιο) Problem

It is an NP-hard problem. Another famous computing problem. You have a knapsack that has a limited weight capacity (you can fill it up to a certain weight). You also have a lot of objects each of which has a value and a weight. The goal is to maximize the value that you fill in the sack, given the weight constraint. In some variants you can take a fraction of an object.

	<p>It is an optimization problem.</p> <p>The naivest approach is the brute force one. Compute all possible combinations and select the best one.</p>
--	--

Systems Design

Intro

Four basic categories

1. Fundamentals of systems: client server architecture, communication protocols etc.
2. Key characteristics of systems: consistency, availability, partition tolerance, redundancy, latency, throughput,
3. Actual components of systems: load balancer, proxy, cache, rate limiter
4. Actual tech of systems: zookeeper, nginx, reddit, S3, etc. (with which you can implement the components to achieve the characteristics you want)

Client server model, some notes

- ✓ First the client does a DNS query
- ✓ Every machine with an IP address (connected to the internet) has a group of 16.000 ports to which programs on that machine can listen to. Usually clients know to which port to send a packet to based on the communication protocol used.
- ✓ Port http 80, https 443, ssh 20, dns 53

- ✓ Have in mind nc for Netcat. It is a command-line utility that reads and writes data across network connections. For example you can listen to one terminal and send an http message to it from another in the same machine.

Network protocols, some notes

	<ul style="list-style-type: none"> ✓ An IP packet has a header and a body. In the header there is a source and destination ip for the packet. <u>A packet is 65KB 2¹⁶ bytes. 1MB needs at least 15 ip packets.</u> ✓ With ip packets only, you can't send a message composed of many packets. There is no standard way to track their order. TCP is a protocol on top of IP and its main purpose when used over IP is to ensure that a message can be composed of many packets and that this message is transmitted reliably. So with TCP over IP you can send reliably, messages of any size. The tcp header is part of the ip packet body.
--	---

- ✓ In order for two machines to communicate over tcp they need first to make a Tcp handshake to create a tcp connection between them. A machine can end a connection or it can timeout on its own.
- ✓ HTTP is build on top of tcp, in order to add some more abstractions to it, specifically the request response communication model. So we forget about ip and tcp packets when we work with http. We only have to deal with http requests and http responses. Http defines a specification for the request and response model. For example the request is an object with various fields that define its behavior (host, port, method, path, headers, body etc.)

Latency and Throughput

- ✓ Latency and Throughput are two of the most important metrics in system design.
- ✓ Throughput is measured in GBps. Latency is measured in seconds.
- ✓ When you design systems you make decisions that affect latency and throughput of your system.

Proxies

A forward proxy acts on behalf of the client. (the server might not know that a request is coming to it from a proxy)

A reverse proxy acts on behalf of the server. (the client doesn't know that its request goes to the reverse proxy first). the reverse proxy can have many applications:

- Filter out certain kinds of requests
- Buffer requests and responses so that even if there is a slow connection with a client, the web server (gunicorn worker process for example) will send the response to the reverse proxy and the proxy will send the response through its buffer. The web server will not block.
- Perform logging
- Cache data
- Act as a load balancer
- Modify request headers
- etc.

Availability

- ✓ Availability describes how resistant is a system to failure, it's fault tolerance. It can be measured in a % of a given amount of time, usually a year, where your service is operational. We typically measure availability with number of 9s. for example 99% availability (or 2 9s) means that your service would be down 3.65 days/year. 5 nines (5 minutes/year) is considered a high standard, it makes a system highly available.
- ✓ High Availability is tough to achieve and it requires some trade offs for example low consistency (me: a db node fails. Data not written in it. a client reads from another db the written data. Another client reads from the recovered failed db the previous data and acts on it for example replying to the old version of a comment. Data is synchronized later. This is eventual consistency vs final consistency.), low latency (me: due to the latency of extra nodes like load balancers) etc. so we should choose carefully which services of our system will be highly available and which will not.

- ✓ The fundamental way to achieve availability is to eliminate Single Points of Failure in your system. This is achieved by adding redundancy. For example you might have more than one servers for a specific service. This is an example of passive redundancy where if one node of the service fails the system continues to work without any modifications. There is also active redundancy. In this case one node of the service is special. It is the one that does the work or handles traffic. If this goes down, one of the others have to take that role (lead election).
- ✓ Another important think that you can do to achieve high availability is to create processes that can handle system failures. For example there might be necessary to have human actions to solve an issue and these humans need to be notified asap.

Maintaining service availability percentages with five-nines requires significant investment and upkeep, using established network configuration, monitoring and troubleshooting networking issues, and following best practices to ensure system components remain operational. Every hour a service is not available can cost a company millions of dollars.

Achieving five-nines availability

How do you get more nines? Consider these steps:

- ✓ Buy the best equipment that's the easiest to repair. Then, add load balancing, failover and redundancy. Highly available systems often include power supplies and processors, battery backup, diesel or natural gas generators for longer power outages than batteries can handle, multiple diverse communication lines and multiples of whatever else is likely to fail.
- ✓ Automate, where possible, to monitor network performance and flag potential malfunctions. Automation tools, along with network analysis software that continually tracks the health of network components and technologies such as AI and machine learning, can help operators reduce the chance for human error and ensure their networks remain operational. Additionally, AI and machine learning platforms can proactively alert network operators in the event of network problems or a security breach and can automatically shift operations from failing components to backups when necessary.
- ✓ Pay attention to software. Out-of-date or unpatched software can make five-nines availability impossible. If a particular component fails because of a faulty OS and takes a long time to get back online, availability will suffer.
- ✓ Test backup and disaster recovery plans to make sure they are sufficient.

Caching

Caching improves latency. In order to cache, you pass the data from a hash function to get a single signature of it and store that data with the hash as the key in a data store. Then you read from there if the key exists.

What happens if you have data stored both in database and the cache and you want to edit it. There are two policies

1. Write through cache policy: we update the cache and then we update the database. One after the other. Sometimes, your Write-Through operations might fail. This can happen due to several reasons like network connection problems or because of an error in your interface implementation logic. Since Write-Through is synchronous, the exception is thrown to the application.
2. Write back (or behind) cache policy: we update the cache only. The database will be updated asynchronously, with a periodic task for example. the application doesn't need to wait for the db write. It gets a success only from the cache. since it is an asynchronous mechanism, failures in write-behind are logged as exceptions and errors in the cache logs but not thrown to the application. it is recommended that you opt for Write-Through if you are dealing with highly sensitive data.

Notice that when you cache mutable data in a distributed cluster of nodes (for example you have a cluster of application servers and you cache data on each one of them), you might have to deal with inconsistency issues (depending on the load balancing algorithm). For example imagine you store youtube comments on individual server caches, and your load balancer distributes requests randomly to the application servers. A user edits his comment, the cached value changes in the server that responded to him, but the change isn't replicated to all server caches. Then another user could read the original message from the cache of another server and reply to the old version of the comment. This is a typical case of inconsistency, or stale data.

One way to solve these issues is by using a separate cache service that all servers use. This could create a single point of failure that you then have to deal with. Another solution might be comment based load balancing with consistent hashing. Requests to the comments service are being hashed consistently, so that each server serves a specific range of comments. This way a comment will always be served from the same server. Depending on the situation you make the necessary decisions. For comments, it is important to have consistency. For other pieces of data, it might not be. For example the view count.

There is also cache eviction policies. Least recently used, least frequently used, fifo, lifo etc. You should pick one based on needs.

Have in mind that in-memory cache is a commonly used approach as I understood. If your data size is not huge and they fit nicely in the memory of your application server, you can just cache it there. Then depending on your needs of course, you can have a simple eviction policy for example clear the cache every 30 minutes.

Client-caching is also commonly used. You can cache in your client's memory so that a client doesn't hit your backend multiple times within the same session.

Load balancer

1. Server-side load balancing
2. Client-side load balancing

Server-side load balancing

A load balancer improves the throughput and latency of your system. It can sit between

- clients and web servers,
- web servers and databases,
- or in the case of dns load balancing, between clients and other load balancers.

In dns load balancing there is a scheme called DNS Round Robin in which a domain name can have multiple IP addresses and whenever there is a dns query for this domain, a load balancer returns one of the ips in a load balanced way.

When new servers are added or removed, we should register or deregister them from the load balancer so that it knows about them.

There are many server selection load balancing algorithms to distribute the load:

1. Random selection
2. Round Robin where servers are selected sequentially from first to last and then back to the first again.
3. Weighted round Robin. If some servers are more powerful than others you can assign increased weights to them so that they receive more load.
4. Performance based selection. The load balancer uses health checks for the connected servers to get some important metrics from them, like average response time, etc. Then it distributes loads to servers that have low response time which could mean that they have less load.
5. Client based selection (based on the IP or username etc.). The balancer hashes the request source IP and routes it to the server that is responsible for handling a range of hashes. This means that requests from the same IP address, will be always routed to the same server. This can be very useful for distributed cache services, where the responses are being cached. This would ensure that a request from the same IP address will be served by the same server which has the cached value in its cache. Notice that to retain the matching between client attribute and servers as much as possible when you add or remove servers, you need to apply consistent hashing or similar.
6. Path based selection. You might route based on the requested path. This way requests to /payments could be served by a specific set of servers and upgrades to other servers will not affect the payments service.

Load balance across regions is achieved with DNS load balancing

You have to apply DNS load balancer to check where a user is and route that request to the region specific load balancer. For example , lets say you have two EC2 instances in Mumbai and another two in Sydney. You can loadbalance between EC2 instances in Mumbai using an ELB deployed in Mumbai region and similarly in Sydney. However to loadbalance between Mumbai and Sydney you will need to use Route 53 and provide DNS name of ELB's in Mumbai and Sydney as endpoints.

Client-side load balancing

Client-side load-balancing: The load balancing decision resides with the client itself. The client can take the help of a naming server (eg. Netflix Eureka) to get the list of registered backend server instances, and then route the request to one of these backend instances using client-side load balancing libraries like Netflix Ribbon.

Advantages of client-side load balancing:

- No more single point of failure as in the case of the traditional load balancer approach.
- Reduced cost as the need for server-side load balancer goes away.
- Less network latency as the client can directly invoke the backend servers removing an extra hop for the load balancer.

Have in mind that you can use “client-side” load balancing for your internal microservices. One microservice is the “client” and the other the “server”.

Hashing

Hashing

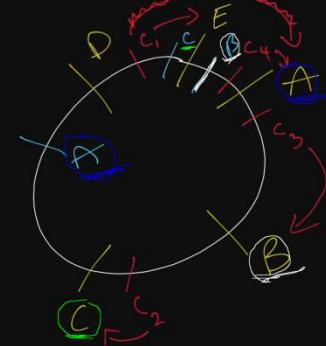
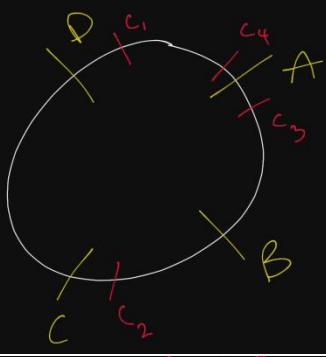
A process that can transform an arbitrary piece of data to fixed size value (usually an integer). It is used by load balancing algorithms

Naive hashing

Assume we use client based hashing for storing client specific cache values in memory of specific servers. A naive implementation of this, would be this. First pass the client ips from a hash function (a hashing algorithm) to get some integers. (Have in mind that the hashing algorithm should provide uniformity. The output values should be uniformly distributed within a range. Some commonly used algorithms that achieve that are MD5, SHA1, Bcrypt). Then you have to assign these integers to 4 servers. A way to do this, is to take the mod of these integers with the number of servers (hashing result integer%4). This would give you values between 0 and 3, so you can map these 4 values to the 4 servers and distribute the load this way.

Notice though the problem: if you add or remove servers you would lose the matching. You would have to repeat the mod operation for the new number and this would result in completely new matching between hash values (ips) and servers. This means that an ip would be served by a different server now, and you will have cache mishits. There are two hashing techniques that can solve this problem of in-memory caching. Consistent hashing and Randevouz hashing.

➤ Consistent hashing



In consistent hashing you don't use the mod operation. You create an abstract circle, within which you assign the servers. To do so, you pass them (their name or something else) through a hashing algorithm. The circle represents the range of the hashing algorithm output. Then you pass the clients through a hashing algorithm (probably the same). This positions them somewhere within the circle. Then you define a rule. A client would be served by the first server it reaches if it moves clock wise.

- This way even if you remove or add a new server, there would be minimum changes on the matching between clients and servers. So you achieve less cache mishits.
- If the hashing algorithm concentrates the servers on one area of the circle, you can pass the servers from multiple hashing algorithms to position them multiple times on the circle.
- You can also add "weights" to specific servers by passing them through hashing algorithms more times than the others. This would position them more times in the circle.

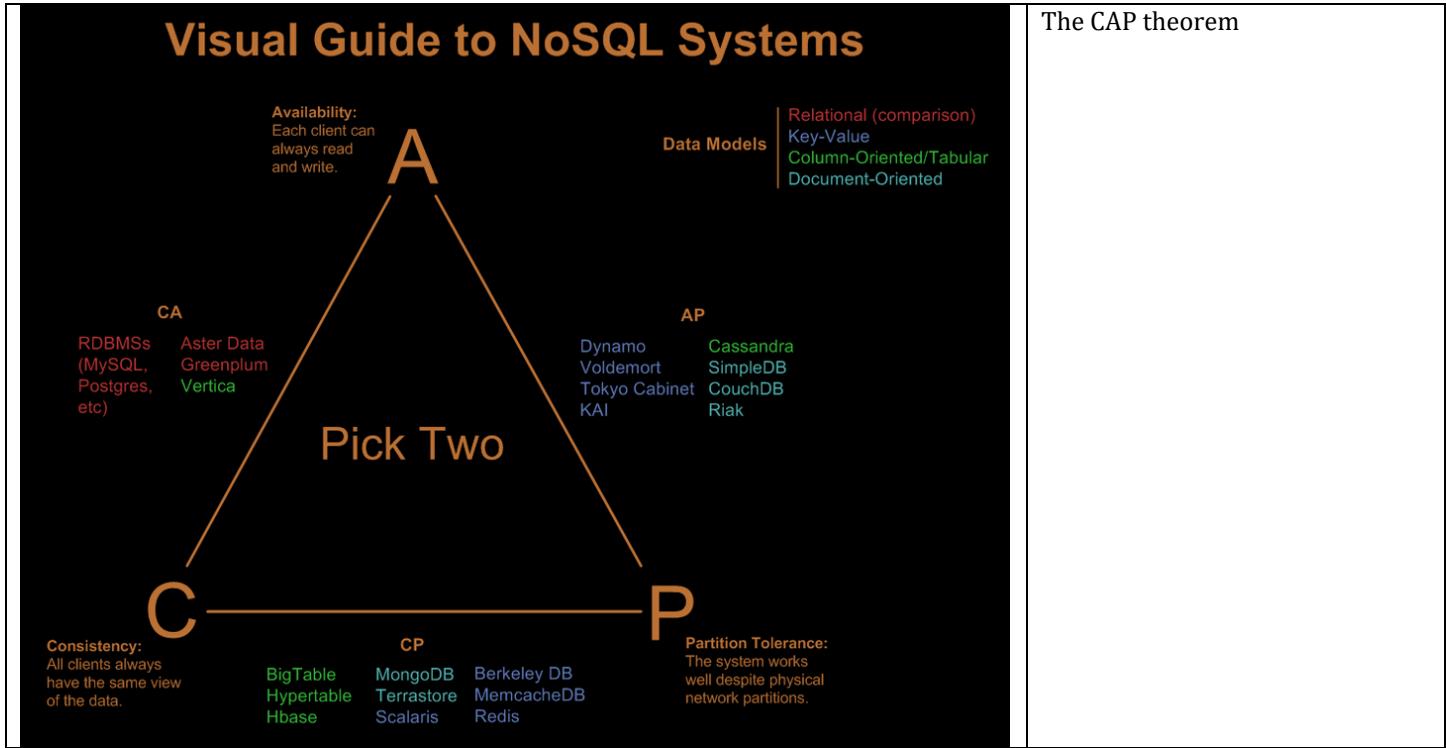
➤ Rendezvous hashing

In this approach you take each client and create a score for all servers. This score is specific to this client. The load balancer assigns this client to the server with the highest score. Each client has a score list with a score for each server. This way if you remove one server, only the clients for which this one was the highest in the ranking would be affected and would need to be assigned to the second highest score.

There are industry grade score computing algorithms that uniformly distribute the values (clients in this case) to the destinations (servers in this case)

Databases

Types of databases



Relational databases

A relational database imposes on the data stored in it a tabular like structure. Data is stored in tables. Tables also called relations. Three of the most fundamental properties of a relational database are their support of:

1. SQL

SQL gives you the possibility to perform powerful queries on the highly structured data of a relational database. Have in mind that if you wanted to perform those queries in let's say python, you would have to load the entire data set in memory and then make the query (using pandas or something). With SQL though this is not required.

2. ACID transactions

A database transaction follows these properties. Atomicity Consistency Isolation Durability. Atomicity: if a transaction is composed of multiple operations (remove money from one account add it to another) then either all operations will be performed or none at all. Consistency: Any transaction respects and abides by the rules of the database. Any transaction sees the effect of all other transactions, there can't be any stale data where one transaction has executed and another doesn't use the results of the executed transaction. Isolation: transactions are executed sequentially, one by one. If a transaction begins after another one, the second one has to wait for the first one to be committed in order to be executed. Durability: the effects of a transaction to the data in the database are permanent.

3. Indexing

This feature increases read speed but decreases write speed (because you have to update the index table too). in a table with 5 million rows, a simple query to get the 10 biggest values, could take more than 15 secs. With an index of that table for this value, it can take less than a second. There are various ways to create an index (bitmap index, dense index, reverse index) but in a nutshell the index is a new table that contains the column of interest sorted, and another column with a foreign key that points to the specific entry that has that value of interest. For example we can create an index for the payment amounts. The index will be sorted relative to the amount, and will have a foreign key to the payments. This way you can search by amount in $O(\log n)$ instead of $O(n)$ or you can get the largest and smallest in constant time $O(1)$.

The CAP theorem

In algoexpert they initially chose a non relational database for some parts of their system. This turned out to be a bad decision. They used google cloud store. One reason was that they couldn't do complex queries on their data. They stored user events on it. They wanted to query that data, for example *select all users that run code within the previous month and they were logged in*. The database didn't support queries with more than one conditions. The other one was that this non relational database offers eventual consistency, which means that some updates didn't reflect immediately on the whole system. They had stale data (data that should but hasn't been updated yet). ACID transactions offer strong consistency. Eventually they chose to migrate to postgresql.

➤ One Index for two columns

You can create indexes from more than one columns. If the two columns are two foreign keys for example, the index would be
1,1->entry1 1,1->entry2, ...

1,2->entry1 1,2->entry2, ...

So you can perform a binary search for combinations of values. For example you want all the 1,1 entries (where both foreign keys are 1).

```
SELECT * FROM sometable WHERE id1=1 AND id2=1;
```

➤ One index per column of the same table

This way you can search efficiently for all indexed columns. For example if your columns 'foo' and 'bar' are indexed both of these queries will be efficient due to binary search:

```
SELECT * FROM sometable WHERE foo='hello';
```

```
SELECT * FROM sometable WHERE bar='world';
```

But what happens in the case in which you search based on both indexed values?

```
SELECT * FROM sometable WHERE foo='hello' AND bar='world';
```

Here the database can make various clever choices on how to most efficiently search. What index to use first for example. If one index returns a much small number of entries than the other, it makes sense to make a binary search for it first, and then sequentially search for the other on the result of the previous search.

Key value stores (Non relational databases)

There are various types of Non relational databases. One of most popular types is the key-value store. It is like a hashmap data structure. The key is a string while the value could be of various types depending on the specific store. It is very suitable for:

1. Cache store

Cached values are stored using the key-value model so they fit perfectly.

2. Dynamic configuration

You have constants that have to be accessible by various parts of your system. You can store these constants in a key-value store. They can be easily modified as well.

3. They are fast

Since data is stored in a hash table like structure, you don't have to search for a specific entry. You just read from the specific key directly. It is O(1) read instead of O(N). so a key value store can increase throughput and decrease latency.

Have in mind that there are many options available, each with its own characteristics in terms of consistency, disk persistence etc.

Specialized Storage Paradigms

➤ Blob stores (S3, GCS from google)

A Blob is an arbitrary piece of unstructured data. Videos, images, large text or binary files are typically considered blobs. It is not convenient to store these pieces of data in a relational database. They are unstructured so they can't be represented with tabular data (the data of a file can't be stored as a group of tables) and they are large. Blob stores are used instead. They follow the key-value model. Each blob has a key.

Bottom line:

- Blob storage is cheap, fast, and just about seven nines of uptime
- Generally well secured, and you have good control over traffic can be shaped
- Great for general storage needs

What you give up is:

- The ability to index the text for fast searching--you might use ElasticSearch as a service to meet that need. (It will work with a lambda function to make sure any new data sent to an S3 bucket triggers an event notification to this Lambda and update the ES index.)
- Acid transactions on the text data

Databases are generally efficient with blob storage up to about 1MB or so, and beyond that most databases recommend storing those blobs outside of the database (Microsoft SQL Server, notably makes this recommendation). You can easily bloat the database making the logistics of replication, backup, and restore more difficult.

➤ Timeseries databases (InfluxDB, Prometheus)

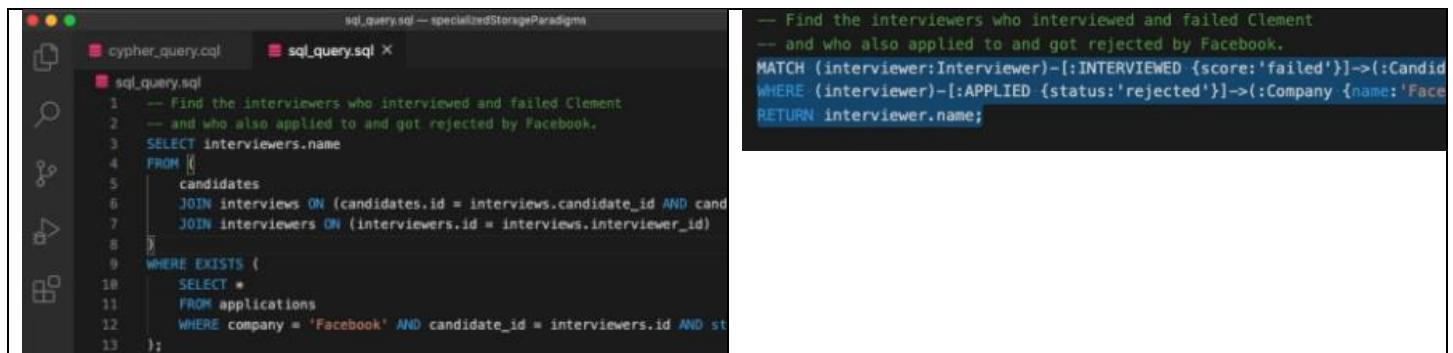
These databases are convenient for storing time-stamped data. For example user events, IOT devices telemetry data, crypto prices etc. If you want to extract various time related metrics like aggregation over time from these data too, then timeseries databases are convenient. Very suitable for monitoring services.

A time series database is built specifically for handling metrics and events or measurements that are time-stamped. A TSDB is optimized for measuring change over time. Properties that make time series data very different than other data workloads are data lifecycle management, summarization, and large range scans of many records.

➤ Graph databases (Neo4j)

There are cases where the dataset contains many relationships between the various data points. For example consider a case where we have people who are interviewers and interviewees in google. Some of them have applied to facebook in the past. You want to find which interviewers who have applied to facebook in the past and failed, have rejected a particular interviewee. In SQL this would require a lot of joins. It would be an expensive query. In neo4j though, it's much simpler and faster. It is like traversing a graph data structure. Neo4j uses Cypher querying language. It also offers a nice interface where you can view your dataset as nodes of a graph.

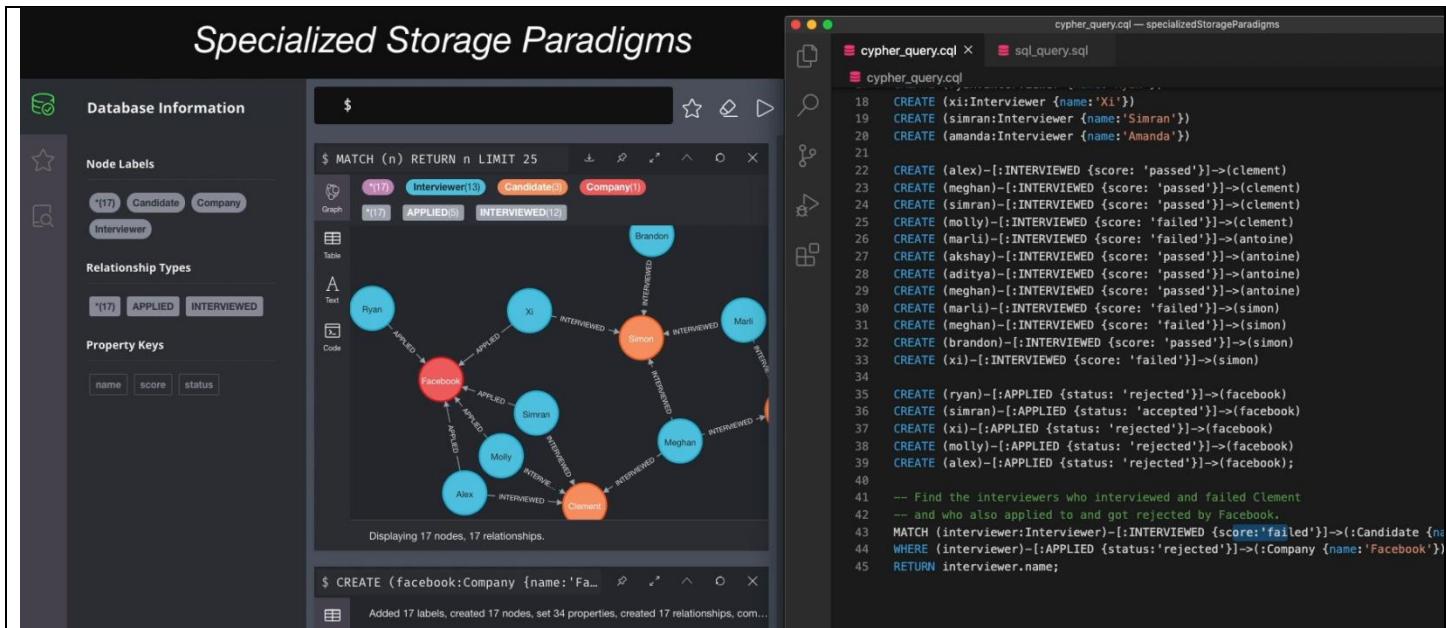
These databases are suitable for social networks which have objects with a lot of relationships.



```
-- Find the interviewers who interviewed and failed Clement
-- and who also applied to and got rejected by Facebook.
MATCH (interviewer:Interviewer)-[:INTERVIEWED {score:'failed'}]->(:Candidate)
WHERE (interviewer)-[:APPLIED {status:'rejected'}]->(:Company {name:'Facebook'})
RETURN interviewer.name;
```

```
sql_query.sql -- specializedStorageParadigms
cypher_query.cql
sql_query.sql
1 -- Find the interviewers who interviewed and failed Clement
2 -- and who also applied to and got rejected by Facebook.
3 SELECT interviewers.name
4 FROM (
5   candidates
6   JOIN interviews ON (candidates.id = interviews.candidate_id AND cand
7   JOIN interviewers ON (interviewers.id = interviews.interviewer_id)
8 )
9 WHERE EXISTS (
10   SELECT *
11   FROM applications
12   WHERE company = 'Facebook' AND candidate_id = interviewers.id AND st
13 );
```

Specialized Storage Paradigms

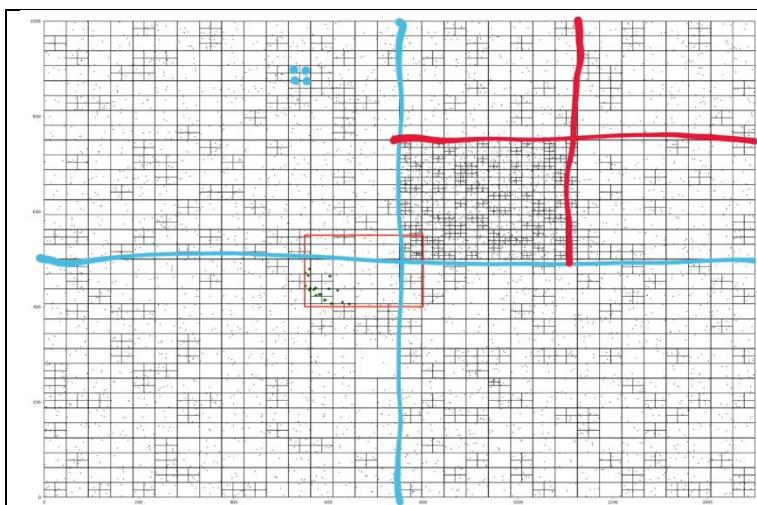


Spatial databases

A spatial database is a database optimized for storing and querying data that represents objects defined in a geometric space. Most spatial databases allow the representation of simple geometric objects such as points, lines and polygons. Some spatial databases handle more complex structures such as 3D objects, topological coverages, linear networks, and TINs (triangulated irregular network). While typical databases have developed to manage various numeric and character types of data, such databases require additional functionality to process spatial data types efficiently, and developers have often added geometry or feature data types.

Spatial databases are optimized for handling spatial data, like longitude and latitude. You perform certain queries very fast, like number of points of interest around a location, line length, polygon area.

To do so, these databases use spatial indexes instead of the regular indexes. There are many types of spatial indexes. A common spatial index is the quad tree. It is a tree where each node has 4 or 0 children.



This is the grid representation of a quad tree. The two axis can have longitude and latitude. The whole grid can represent a specific area or the entire globe. All points of the globe are located within this grid. The initial rectangle represents the root node of the tree (the root node of the tree represents the entire globe). We split the rectangle in 4 parts. Each part represents a child of the root node. The process goes on. There is a criterion which says that we stop dividing when the number of points inside a rectangle becomes less than a limit. This means that densely populated areas will be represented with a dense grid.

Now when we search for a specific location (long and lat) we compare it with the 4 first children to determine in which one it belongs and we repeat in the next level. This means that in one operation we eliminated 3/4 of the tree. Which means that searching a quad tree is a $\log_4(n)$ which is even better than $\log_2(n)$.

Me: the quad tree is a tree so it is stored like a tree. It has a value and first, second, third, fourth child which reference the children node memory slot. Or it could be stored as an array. The value can be the long, lat pair. The point you want to search for is another long, lat pair. You calculate the Euclidean distance between it and the 4 nodes and select the closest one. This way you do 4 distance calculations, but then you eliminate the 3 nodes so there is no need to search in them, but only on the selected one. You repeat the process until your criteria are met.

Table partition

Most sql dbs offer table artition functionality. This is necessary when you have to deal with very large tables with millions of rows. The concept is that you split the table either horizontally (rows based) or vertically (columns based) to smaller tables so the query is executed against a small subset of the whole data and is much more efficient.

Typical example could be user following table where you have follower_id and followee_id. If you have billions of users, this table would be huge. Initially you should think if there is a way to avoid the large table all together. For example you could have a follower-count int field, in the user profile, and a followers-list json field to store the actual followers. This would increase the write burden though.

To make the partition you use a partition key. In the followers example, you could use as a partition key the “follower-id” field, to make the partition. for example follower-ids 1to100k go to table 1 etc. when there is a search for user-id 900,001, we identify the respective partition table and search there. This is called range partitioning. There is list partitioning for example partition based on the value of a field, all rows with country=Paris go to one partition etc. There is also hash partitioning which is a pseudo random breakdown of your data.

You can have vertical partitioning too. for example you can have a blob field (large binary file) in your user profile table that contains some documents in binary format. You could partition the table vertically and store the blob column in another table.

Partitioning is transparent to the client. The logic to determine the partition table to query is not in the client but in the database itself.

- You will typically start to see the performance benefits with tables of 1 million or more records, but the technical complexity usually doesn't pay off unless you're dealing with hundreds of gigabytes of data.
- Query on 1.3m rows took 4.1 secs. On the partitioned in two table it took 1.23 secs. If the rows were 500m the difference would be significant.
- The partition key must be a primary key?
- Django doesn't support partitions out of the box for postgresql, so you have to use a 3rd part library (Django-postgres-extra)

In very large systems, you could have many shards, and then partitioning per shard and of course index per partition.

Replication

Replication offers

- Resiliency. You can implement a master slave architecture. If your master goes down your slave takes charge.
- Performance boost. You can create write and read replicas. So heavy writes don't affect reads. You can have location based replicas so that local queries are handled faster by local instances (and then the changes are synchronized to the other replicas).

Replication vs sharding

Sharding is another approach for increasing queries performance.

You can use database replication to avoid down time for your system that can be caused by your database server failing. Essentially you have your main database and a replica. Notice that there are two modes of operation for the replica system. Keep the replica updated synchronously or asynchronously.

- Synchronous synchronization (final consistency)

In the first case, when a write is to be made to the main database, it should also be executed in the replica afterwards in an ACID like way. If the write on the replica fails, the whole write operation fails too. This way the two databases are always in sync with each other. This means that the system's availability is increased since if the main database goes down the replica can take over. But it increases latency since a write should be completed in two different databases. Synchronous synchronization increases availability but increases latency too. have in mind that you must apply readlock to the read dbs when a write is done to the write db. When this write is successfully synchronized then the readlock can be removed.

- Asynchronous synchronization (eventual consistency)

In the second case, the replica is updated asynchronously. For example in a periodic way, every 5 minutes or so. In this case the latency of the system is better (lower) in relation with a synchronous system but it might be less consistent. If something happens to the main database before the replica is updated then the replica would take over with the old version of the data. Asynchronous synchronization decreases latency but decreases consistency too. In general, all systems that use eventual consistency need some kind of conflict resolution (for example the last update wins) because two clients connected with two different nodes might change the same row simultaneously. A server-side stored procedure can be used to define what must be done if a conflict occurs.

Depending on your case, you can choose a strategy. If you don't care to have non instant updates you can use the async synchronization. This might be the case when you want to populate a feed. A post doesn't have to be instantly visible to the feed. It might take some time to appear without greatly affecting the quality of the feed. If you have users in america and asia and have two databases, one in us and one in korea, that act as the main database for the region. So a post of an american user would be written in the us database will not be visible to the asian users feed until the asian database is updated asynchronously.

You can have different combinations of write and read architectures.

- One write db and multiple read dbs where read dbs are regional dbs (reads are routed by location via a load balancer).
- Regional write and read dbs. In this case the writes are also routed by location. And the write dbs need to be in sync (or not depending on the case). An american user moving to asia, should not lose his follows etc. if he logins. If the app is not business critical you can have eventual consistency between the regional dbs. Have in mind that this is not sharding. It is replication.

Feed generation

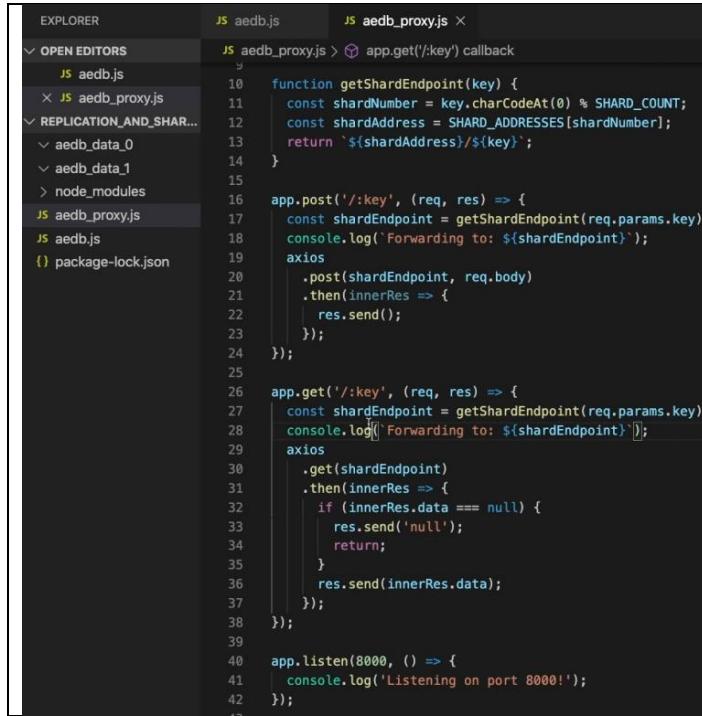
Usually, calculating the feed for a user, is a computationally expensive operation. It might include ML algorithms to run on user data to get his interests and collect relevant posts, popular posts, proposed posts etc. so it is better to precompute a feed for each user let's say every hour, and store it in the cache layer.

Bidirectional replication

Have this possibility in mind. Bidirectional transactional replication is a specific transactional replication topology that allows two servers to exchange changes with each other. For example for SQL server: each server publishes data and then subscribes to a publication with the same data from the other server. The @loopback_detection parameter of sp_addsubscription (Transact-SQL) is set to TRUE to ensure that changes are only sent to the Subscriber and do not result in the change being sent back to the Publisher.

Sharding

In sharding you distribute the data to different databases. So each database has a portion of the data. It's up to you to decide for the way the data is being distributed among shards. For example you might choose to have the same schema in all databases, and distribute the rows of the tables. So each database would have a portion of the users, for example. A good way to distribute the data is using hashing. (another way is to distribute the data based on location, for example based on the zip code of an entity). Consistent hashing could be an approach. It works well if new databases are added but it doesn't protect us from a database going down. That's why you should have replicas for each shard. Notice that if your hashing function is not a uniform one, you might end up with a hot spot (an overloaded shard).



```

EXPLORER JS aedb.js JS aedb_proxy.js
OPEN EDITORS JS aedb.js JS aedb_proxy.js > app.get('/:key') callback
REPLICATION_AND_SHAR... aedb_data_0 aedb_data_1 node_modules aedb_proxy.js aedb.js package-lock.json
10  function getShardEndpoint(key) {
11    const shardNumber = key.charCodeAt(0) % SHARD_COUNT;
12    const shardAddress = SHARD_ADDRESSES[shardNumber];
13    return `${shardAddress}/${key}`;
14  }
15
16  app.post('/:key', (req, res) => {
17    const shardEndpoint = getShardEndpoint(req.params.key);
18    console.log(`Forwarding to: ${shardEndpoint}`);
19    axios
20      .post(shardEndpoint, req.body)
21      .then(innerRes => {
22        res.send();
23      });
24  });
25
26  app.get('/:key', (req, res) => {
27    const shardEndpoint = getShardEndpoint(req.params.key);
28    console.log(`Forwarding to: ${shardEndpoint}`);
29    axios
30      .get(shardEndpoint)
31      .then(innerRes => {
32        if (innerRes.data === null) {
33          res.send('null');
34          return;
35        }
36        res.send(innerRes.data);
37      });
38  });
39
40  app.listen(8000, () => {
41    console.log('Listening on port 8000!');
42  });
43

```

Have in mind that the logic of distributing the data to shards can live in your application server but it is common to be in a separate service, a reverse proxy to which the application server speaks to, to read and write data, and the proxy applies the logic. This is a simple reverse proxy that distributes the data based on a simple logic in getShardEndpoint.

Notice

I am well aware that sharding should probably be used only as a last resort optimization since it has limitations and really makes things quite complex. Most people don't need sharding: an optimized master/slave architecture can go a very long way. Have in mind django's automatic database routing features.

You should shard only when it is absolutely necessary

0. Understand what your actual problem is before optimizing(too slow reads vs too slow writes) Analyze your slowest queriers and see why its slow. Create indexes on appropriate columns and tune your data schema.
1. Horizontal Partitioning - Have partition key(mostly on primary key) and split database into different ranges. This will create smaller B-trees on the indexes.
2. Vertical Partitioning - When you have columns that you rarely access, and you cut a column out of the main database. This will make reads faster for frequent queries and slower for not frequent queries and make your B-trees smaller(less space in memory also)

3. (Read Optimization) Master-Slaves replications, with master handling all writes and slaves handling all reads. Masters sync with slaves with latest data

4. (Write optimization) Master-Master-Slaves replication in multi-regions(i.e. EAST vs WEST) with multiple masters handling writes, slaves handling reads, sync handled between masters<->masters and masters->read

5. Finally shard when you REALLY have to. Sharding adds complex client logic and couples with your data base. You also lose ACID as you cannot guarantee transactions and isolation between your shards. You can use Vitess to help you manage your sharding query logic

Tips

CAP theorem

In theoretical computer science, the CAP theorem, also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

- Consistency: Every read receives the most recent write or an error
- Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability.

The following data types are supported by PostgreSQL:

- Boolean
- Character Types [such as char, varchar, and text]
- Numeric Types [such as integer and floating-point number: float, float8, numeric(digits, decimals)]
- Temporal Types [such as date, time, timestamp, timestamptz, and interval (storing periods of time)]
- UUID [for storing UUID (Universally Unique Identifiers)]
- Array [for storing array strings, numbers, etc.]
- JSON [stores JSON data, json and jsonb]
- hstore [stores key-value pair]
- Special Types [such as network address: inet (ipv4,6), cidr, macaddr and geometric data]
- Text search types

PostgreSQL provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language documents to locate those that best match a query.

- Composite types
A composite type represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.
- Range types
Range types are data types representing a range of values of some element type (called the range's subtype). For instance, ranges of timestamp might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is tsrange (short for “timestamp range”), and timestamp is the subtype.
- Pseudo types (for example serial)

CHAR is a fixed length field; VARCHAR is a variable length field.

UUID

The UUID data type allows you to store Universal Unique Identifiers defined by RFC 4122. The UUID values guarantee a better uniqueness than SERIAL and can be used to hide sensitive data exposed to the public such as values of id in URL. This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm. Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

The UUID stands for Unique Universal Identifiers. These are used to give a unique ID to a data that is unique throughout the database. The UUID data type are used to store UUID of the data defined by RFC 4122. These are generally used to protect sensitive data like credit card informations and is better compared to SERIAL data type in the context of uniqueness.

In Django whenever we create any new model, there is an ID-or PK, model field attached to it. The ID field's data type is integer by default. To make this integer id field UUID, we can use UUIDField model field, which was added in Django 1.8

SERIAL pseudo type (sequence)

When creating a new table, the sequence can be created through the SERIAL pseudo-type as follows:

```
CREATE TABLE table_name(
    id SERIAL
);
```

This is equivalent to this:

```
CREATE SEQUENCE table_name_id_seq;

CREATE TABLE table_name (
    id integer NOT NULL DEFAULT nextval('table_name_id_seq')
);

ALTER SEQUENCE table_name_id_seq
OWNED BY table_name.id;
```

You create a sequence using the SERIAL pseudo type. (this is the default mechanism for primary key in django)

In PostgreSQL, a sequence is a special kind of database object that generates a sequence of integers. A sequence is often used as the primary key column in a table.

It is a pseudo type because the actual type is integer and there is a sequence object attached to it. it is not a real data type.

Range types

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

Every range type has a corresponding multirange type. A multirange is an ordered list of non-contiguous, non-empty, non-null ranges. Most range operators also work on multiranges, and they have a few functions of their own.

NoSQL databases

Use cases

Basically, it's rarely the question of one or the other. A SQL database will usually be the backbone of on any given application, but a NoSQL database will come into play to solve specific problems where performance and scale pose the main issue.

- Store aggregated (report) data from expensive sql queries so that you can access them quickly and do basic querying on top of them too if you want. A typical case would be to store a report from data stored in many tables. You perform the expensive joins to create the report data, and then store it in a no sql db like mongo from which you can read it quickly. It could be the case that you made your application so that 90% of your client's reads, are served from nosql aggregated data (for example from elasticsearch) while writes go to the sql data.
- Store performance critical data that has rigorous and pointed access patterns (meaning that it needs no further joins). In applications like gaming, social media, IOT. For example, keep track of users in a SQL database, keep track of live game attributes (of a live round of a game) in a NoSQL database. Assuming your game is designed correctly, you'll be able to fetch and update such

attributes very quickly and for many players, and since your games access patterns should be very predictable, you won't have to deal with a situation where you'd need to perform a join or something on that data. Such scale/performance may not be achievable with a SQL database.

- For storing messages, time-series, dumps, raw data, logs, documents, file store metadata and so on. These are applications that do not have as much relational data or where relational data is computed/referenced.

Schema on read vs Schema on write

With NoSQL you have "schema on read" where you delegate the schema construction into your application's data layer (that consumes the data stored in nosql), as opposed to "schema on write" where you delegate the schema construction into the database when you store the data. Have in mind that when you consume noSQL data, you have to deal with deserializing, parse errors, data version incompatibility, and a whole host of other problems that SQL databases have already been solving for 40 years.

Normalization in Relational dbs tries to achieve data consistency by avoiding duplicated data (and consequently it minimizes the space data occupies in the disk). NoSQL kind of says "to hell with saving disk space. If a doctor has multiple patients, and patients see multiple doctors, we'll just duplicate the doctor data under each patient and duplicate patient data under each doctor". It can vastly improve response times at the cost of data consistency (and disk space). Once you want to update the patient daily medicine you will need to update it in 20 different places

Use cases

 Scott McNulty · Follow NoSQL DBA, 7 Years. Relational DBA, 25 Years - 5y When you have a fairly narrow variety of queries. It's also good when your data doesn't change frequently. When the records have to remain "whole" rather than being normalized. When you are in the cloud and are distributing copies of your records for redundancy. When you need a db and you have plenty of programming knowledge but not in SQL. Situations where you need data to propagate but some slowness could be okay. Anything where joins are expensive. Anything where someone built a materialized view. Also when you want to replicate the data from an RDBMS for some specific purpose. You can run queries and convert the results to a JSON record to be sent to MongoDB. For example: <ul style="list-style-type: none">• Catalogs - lots of reads, not very many writes• Archive of records - no reason they have to stay in their native db when they may only be kept for legal or emergency reasons.• Student tests - pull the whole test from mongo, tabulate the answers in a different kind of db.• Content management - well they are documents and it is a document db.• Log data or Event stream data - you could use sql server, but why bother? Mongo even has a Time To Live feature that can remove older information.• Daily Reports - you can store them in MongoDB and call up a given one at a moment's notice if you've built the system right.• Completed Invoices - you can store a copy of these and use that to pull from rather than regenerating them from your RDBMS.• User/Password data - you store the user name and hash together in a document.• Cloud Database - SQL Server still has issues in the cloud, last I heard. Note that I've listed cases where MongoDB is good at standing alone and also cases where it hybridizes well with another db.	... Have in mind that postgresql as well as other RDBMS have implemented a json field to which you can store json strings. It's not just a string field, it's fully searchable. So if you just want to store json, you can use that instead of a no sql db. Django doesn't play very well with mongo. There are some 3 rd party Django-mongo bridges that you have to rely on. Fast api and flask though, work very well with mongo.
--	---

Mongodb usecases

- Customer Analytics

Creating consistently good customer experiences has become a key challenge for many organizations. The reality is that our expectations around what a good customer experience is has increased dramatically over the past few years. What used to be cool and different is now the norm.

Data aggregation is one of the keys to creating amazing customer experiences. Companies are collecting massive amounts of data about their existing and potential customers and aggregating it with publicly available data. This data can tell companies how customers interact with products (digitally and in person), personal preferences, demographics, etc. From all of this disparate data, companies can build customer profiles and nurture paths with the goal of getting the customer to buy more products.

With all of this data coming from different sources with different schemas, tying it all together at such a massive scale is a huge challenge. The flexibility and scalability of MongoDB provides a solution. MongoDB allows for the aggregation of this data and building analytical tools in order to create amazing customer experiences. MongoDB's speed allows for dynamic experiences that can evolve based upon the customer behavior in real time.

- Product Catalog

Product catalogs are not new to the evolving digital experience. What is new is the volume and richness of the data that feeds the interactions in product catalogs that we use today. MongoDB provides a great tool to store many different types of objects with different sets of attributes. MongoDB's dynamic schema capability allows for product documents to only contain attributes that are relevant to that product. Gone are the days of needing every product record to contain every possible attribute. MongoDB users can very quickly and easily make changes to their catalogs, providing a better experience for developers and customers.

- Real Time Data Integration

Companies have vast amounts of data spread across their organization. Data provides value if it's aggregated in one "single view". Previously, energy and resources were spent on data ingestion, transformation, and schema changes in order to obtain a single source of data. MongoDB's flexibility and query capabilities make it easy to aggregate this data and create the tools that make organizations more efficient. This aggregation can be achieved to provide a "single view" of their data in real time. With the addition of change streams in MongoDB 3.6, developers can now monitor and take action on specific events quickly.

- Mobility and Scaling

With most mobile application development, companies are dealing with varying data structures coming from multiple sources and potentially highly dynamic growth. The flexibility and scalability of MongoDB provides a great database solution for dealing with this type of environment. With schemas that can evolve over time, mobile application developers don't have to spend time adjusting the database. Instead, developers can focus on developing the customer experience.

Intro

NoSQL databases can be categorized based on the data model types (the way they store data). Types of NoSQL dbs

1. **Key-Value** databases (Cassandra, Redis, Amazon DynamoDB –cloud, Riak etc),
2. **Key-Document** (MongoDB),
3. **Column-Family** (Cassandra, HBase, BigTable) and
4. **Graph** (Neo4j, OrientDB).

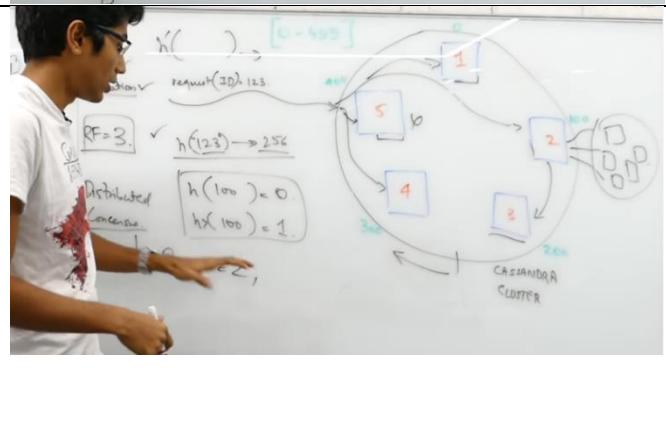
We've heard it before, relational databases don't scale, but NoSQL databases do. Many NoSQL systems were built explicitly with scale in mind so you'd expect them to scale. But they have made major concessions... If you don't need the following relational features, then NoSQL may work really well:

- Constraints and referential integrity
- Transactional guarantees
- Ability to easily analyze your data (SQL)

If you started with NoSQL and don't need any of the above you should be in great shape. If you started with a relational database and now need scale, well then NoSQL can help if you 1. don't have the above requirements and 2. can invest in all the application changes needed to detangle your relational model.

Nosql databases are used as clusters, for example cassandra cluster, mongoDB cluster etc. It is this architecture that offers them many of their properties both good and bad. For example they have inherent redundancy but consistency is not guaranteed

since the same data is copied in multiple nodes. This means that there are edge cases where there would be inconsistent versions of the data in different nodes.



Advantages

- Build for reading an entire data blob in one operation (no joins). Built for inserting and retrieving all data with a single operation. So you avoid table joins which can be very expensive.
- No need for schema. Built for flexibility. The database doesn't care about a schema, all it cares about is a json document. For example if the address of the user is missing you just ignore it completely and store the rest.
- Built in horizontal partitioning. Built for scalability. They transparently apply horizontal partitioning (sharding) of the data.
- Built for aggregations (analytics applications). Some are better than others.

Disadvantages

- ACID is not guaranteed. Not built for regular updates. Consistency is not guaranteed. Two nodes might have different data for the same key. Also you can't make database transactions.
- Not optimized for reading specific data. If you only want the ages of your employees then for each key the whole blob of data must be read and then filtered for the age attribute.
- Relational constraints can't be enforced. No implicit information about relations. For example a row with a foreign key can exist only if the entry the foreign key points to exists.
- Joins are not efficient. You have to read all the blob data, filter for the joined attribute and combine. You do this manually.

An example of a nosql database architecture

You have a cluster of nodes that works as a sharded system. The operation requests are hashed to a key and each node handles a certain key range. In order to handle node failures that might lead to data losses there are replications between nodes. For example in cluster of five nodes you can have a replication factor RF of three. This means that the data of each node is replicated to additional two nodes. Since the same data lives in more than one nodes there is a distributed consensus between these nodes to decide if the data exists. For example you can have a quorum of two as a consensus mechanism. This means that from the three replicated nodes the two must agree. When the data is written to the first node it is concurrently written to the second and third one. If the first node dies after writing, data might or might not be written in the others. If they aren't then if a read request comes for that data it will be responded either with a database error (if 3/3 quorum) or with no data (2/3 quorum). This is a choice of availability over consistency. The database remains available despite a crash but the data is not consistent for a while and the users might notice it. It is a quite rare case though.

Aggregation in nosql databases

Not all are optimized for aggregations

MongoDB is not optimized for aggregation. Given the same data, there are lots of other DBs that could run those aggregations faster. But those DBs make other tradeoffs. In mongo Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

RavenDB aggregates data from its documents using MapReduce. The first time an aggregation is needed all data are read and the needed index for them is created. This is an expensive operation but is done only once. The aggregates will be automatically recomputed every time a new entry is added to the nosql database.

What if you want to store users and items for example? You should find a proper way to create the keys since ids will not do (user:user-id and item:item-id would do I guess).

A "Key-Value database" defines that the "Value" part of the data to retrieve can be accessed directly by querying the database for its "Key", as opposed to define database schemas and querying the database using SQL syntax. The "Value" could be anything, a simple string, a programming language object, an HTML page...

Document database says more about the "Value" part of the Key-Value pair: the data retrieved is encoded in some standard, portable format, as JSON or XML. This gives the database some structure, but nowhere near as rigid as a traditional database where every row has to conform to the defined schema.

MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON

NoSQL vs relational dbs

In a traditional RDBMS, the basic operations are reads and writes. Reads may be scaled by replicating data to multiple machines, thus load-balancing read requests. However, this affects writes because data consistency must be maintained. Writes only may be scaled by partitioning the data. This affects reads, as distributed joins are usually slow and hard to implement. Additionally, to maintain ACID properties (Atomicity, Consistency, Isolation, Durability), databases must lock data. This means that when one user opens a data item, no other user should be able to make changes to the same item. This restriction has serious implications on performance. These limitations have not been a major problem in the past. However, with the advent of social networking and big data, a number of massive databases that emerged were forced to serve the tens, or even hundreds, of millions of clients throughout the world with several thousand reads and writes every minute. Traditional RDBMSs simply do not meet this need because they can only "scale up," or increase the resources on a central server. A NoSQL implementation, on the other hand, can "scale out," or distribute the database load across more servers. NoSQL databases offer the advantages of rapid scalability, much better performance, and a simpler structure compared to RDBMSs. However, they also suffer from being a relatively new and unproven technology, and they cannot provide RDBMS' rich reporting and analytical functionality.

They can't do efficient Joins and they don't handle normalized data efficiently. You have to de-normalize your data as much as possible.

Misc

Good one. Why you don't want Mongo with django

<https://www.pydanny.com/when-to-use-mongodb-with-django.html>

Mongo

A Collection in mongo is like a table in a relational database. It holds documents. A document which is a json string is like a table entry.

RDBMS – Mongo notation

(Database – index) no sure if this is how a db like entity is called in mongo.

Table – Collection

Row – document (json string)

Column – a json attribute

Cassandra

column-oriented database

This means that the columns of the typical relational database, become rows in cassandra. This feature along with its distributed model made it popular. Initially developed by facebook.

Cassandra is a column-oriented database, meaning that its rows actually contain what we most usually think of as vertical data, or what is traditionally held in relational columns (Rows contains columns? Huh?). The advantage of column-oriented database design is that some types of data lookups can become very fast, given that the desired data could be stored consecutively in a single row (compare this with having to search and read from multiple, nonconsecutive rows to attain the same field value in row-oriented database). This particularity, along with the optimized, decentralized distributed model has solidified Cassandra's popularity over the years.

Amazon DynamoDB is a key-value and document-oriented store, while Apache Cassandra is a column-oriented data store.

Mongo vs elasticsearch

Elasticsearch is built for search and provides advanced data indexing capabilities. For data analysis, it operates alongside Kibana, and Logstash to form the ELK stack. MongoDB is an open-source NoSQL database management program, which can be used to manage large amounts of data in a distributed architecture.

MongoDB

Mongo favors consistency (single master)

Cassandra favors availability (no master)

One of the most significant differences between MongoDB and Cassandra is their strategy concerning data availability.

- MongoDB has a single master directing multiple slave nodes. Although the strategy of automatic failover does ensure recovery, it may take up to a minute for the slave to become the master. During this time, the database isn't able to respond to requests.
- Cassandra, on the other hand, uses a different model. Instead of having one master node, it utilizes multiple masters inside a cluster. With multiple masters present, there is no fear of any downtime. The redundant model ensures high availability at all times.

Mongo uses a single master node to which all writes are going. If it goes down then you have some time of unavailability until a new master is elected. A group of one master along with its secondaries is called a replica set. When you shard you have many replica sets. Cassandra has no master node and applies eventual consistency. It also offers tunable consistency for each query you specify the quorum of nodes that need to agree on the data. Keyspace is like a database.



Replica Set Quirks

- A majority of the servers in your set must agree on the primary
 - Even numbers of servers (like 2) don't work well
- Don't want to spend money on 3 servers? You can set up an 'arbiter' node
 - But only one
- Apps must know about enough servers in the replica set to be able to reach one and learn who's primary
- Replicas only address durability, not your ability to scale
 - Well, unless you can take advantage of reading from secondaries - which generally isn't recommended
 - And your DB will still go into read-only mode for a bit while a new primary is elected
- Delayed secondaries can be set up as insurance against people doing dumb things

Sharding Quirks

- Auto-sharding sometimes doesn't work
 - Split storms, mongos processes restarted too often
- You must have 3 config servers
 - And if any one goes down, your DB is down
 - This is on top of the single-master design of replica sets
- MongoDB's loose document model can be at odds with effective sharding

Neat Things About MongoDB

- It's not just a NoSQL database - very flexible document model
- Shell is a full JavaScript interpreter
- Supports many indices
 - But only one can be used for sharding
 - More than 2-3 are still discouraged
 - Full-text indices for text searches
 - Spatial indices
- Built-in aggregation capabilities, MapReduce, GridFS
 - For some applications you might not need Hadoop at all
 - But MongoDB still integrates with Hadoop, Spark, and most languages
- A SQL connector is available
 - But MongoDB still isn't designed for joins and normalized data really.

Database – table – row

Database – collection – document

_id fields is automatically assigned to every document you put into mongo

No fast lookups if not the same schema in all documents (because no index?)

Also if you want to shard, the document model must have some unique key to shard based on it.

MongoDB	SQL
Document	Row
Field	Column
Collection (of documents)	Table
The database is a set of collections	The database is a set of tables

Looks like JSON. Example:

```
{  
    "_id" : ObjectId("7b33e366ae32223aee34fd3"),  
    "title" : "A blog post about MongoDB",  
    "content" : "This is a blog post about MongoDB",  
    "comments": [  
        {  
            "name" : "Frank",  
            "email" : fkane@sundog-soft.com,  
            "content" : "This is the best article ever written!"  
            "rating" : 1  
        }  
    ]  
}
```

You can't reference data on a collection in another database. Collections need to be in the same database.

If you want to store recommendations you might prefer the availability of Cassandra. If you want to store financial data you might want the consistency of mongo.

Multi document ACID transactions

Mongo offers this since 2018 (v4.0). MongoDB transactions work similarly to transactions in other databases. To use a transaction, start a MongoDB session through a driver. Then, use that session to execute your group of database operations. You can run any of the CRUD (create, read, update, and delete) operations across multiple documents, multiple collections, and multiple shards.

Have in mind that you

Misc

Have in mind that you can do these things with a nosql db too (Mongo)

- You can start a MongoDB database with your documents in individual collections/tables just like SQL and performs joins using the aggregation framework. If you find that a frequently used query is slow, you can then just create an index for that query. If that is still too slow, you can then start looking at moving those documents into subdocuments and generally denormalizing your data more heavily to suit your use case.
- You can also guarantee multi collection/table ACID consistency using transactions on the write side.

But!

The end result is that the MongoDB queries are 1 to 2 magnitudes slower than the PostgreSQL queries because of limitations on how the joins can be structured and the flexibility of the data. And that is on top of dealing with data inconsistency and missing key features like foreign key restrictions and similar data-consistency tools which have long been available in any relational database.

Cassandra

Cassandra favors availability

Cassandra has no master node and applies eventual consistency. It also offers tunable consistency for each query you specify the quorum of nodes that need to agree on the data. Actually this seems like an amazing feature. Although you don't avoid eventual consistency problems.

<https://www.yugabyte.com/blog/apache-cassandra-lightweight-transactions-secondary-indexes-tunable-consistency/>

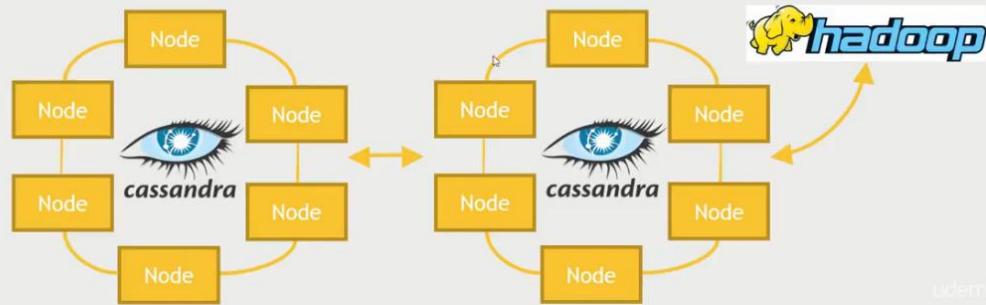
Keyspace is like a database.

It offers a CQL language for querying data. It's not sql, no possibility for joins. All

Cassandra works in a ring architecture with no master. It uses a gossip protocol for communication between nodes.

Nodes can be spread across datacenters.

- Cassandra's great for fast access to rows of information
- Get the best of both worlds - replicate Cassandra to another ring that is used for analytics and Spark integration



In this example you have a Cassandra cluster that serves your application and a copy of this same cluster which is used with Hadoop for analytics. This way you don't put additional pressure to your application cluster. There are opensource connectors between Cassandra and spark like datastax. You interact with a hadoop dataframe the data of which is stored in Cassandra.

CQL (Wait, I thought this was NoSQL!)

- Cassandra's API is CQL, which makes it easy to look like existing database drivers to applications.
- CQL is like SQL, but with some big limitations!
 - NO JOINS
 - Your data must be de-normalized
 - So, it's still non-relational
 - All queries must be on some primary key
 - Secondary indices are supported, but...
- CQLSH can be used on the command line to create tables, etc.
- All tables must be in a *keyspace* - keyspaces are like databases

Lead election

Lead election is a process that assures that an operation is performed only one time (by the leader).

The problem of leader election is for each processor eventually to decide whether it is a leader or not, subject to the constraint that exactly one processor decides that it is the leader.[3] An algorithm solves the leader election problem if:

- States of processors are divided into elected and not-elected states. Once elected, it remains as elected (similarly if not elected).
- In every execution, exactly one processor becomes elected and the rest determine that they are not elected.

While the lead election mechanism seems like an alternative of a workers queue where the first worker that will get the work from the queue will execute it, the problem with the asynchronous job execution system (the queue model) is that it is at least once or at most once delivery systems. On the contrary leader election is an exactly once system (right?). Have in mind that exactly-once delivery is one of the hardest problems to solve in distributed systems.

Imagine you want to implement a subscription based service. You have a database in which you store the subscription details, when the user subscribed, the amount that will be charged, when it will be charged next etc. The actual payment is done by a third party service. You can have a distinct service that communicates with your database (let's say periodically) and then speaks to the 3rd party service if there is a payment to be processed.

If you want to add redundancy to this service you can run it in more than one servers. Notice though that only one of them has to communicate with the 3rd party service to initiate a payment because the payment should be done only one time. (I think that a queue with workers would be a good solution for this case, but it can also be done with lead election).

Lead election is a process in which a group of distributed nodes, agree on which of them is the leader (that can then run some leader specific logic). selecting a leader is a very difficult problem in computer science. It requires a group of distributed nodes to reach to a consensus about a certain value. There are industry grade consensus algorithms used for this purpose, Paxos and Raft are two of them.

Actually you use 3rd party services that have already implemented these algorithms. Services like apache [zookeeper](#) and [etcd](#), they can also do lead election among others so you can use them.

Etcd is a key value store that is highly available and strongly consistent. This combination of the last two, is very unique. Etcd achieves that by using lead election implementing the Raft consensus algorithm. Etcd runs in a group of nodes itself (since it wants to provide high availability), but it has only one data store (can the data store be sharded and how does this affect etcd properties). When you write to the etcd data store, its nodes elect a leader that will do the write to the main data store. So the data store is always at the latest state (either a value has been stored or not there is no partial storing in some nodes, but there is only one data store node, or is it?). This way you have both high availability and strong consistency.

Back in our case, we can use these properties of etcd to achieve leader election in our system. Actually we will store a “leader” key in the etcd data store, the value of which will be the leader of our group. Instead of implementing a consensus algorithm between our nodes to agree on a leader, our nodes will read the leader from etcd. Initially all of them will try to write their name on this key, but etcd only allows to write to this key if it doesn’t already exists (or if it has expired). the node that will write its name to this key first will be the leader. Then it has to keep refreshing it’s leadership until it can’t for some reason. Essentially we get who the leader is by trying to insert this key. In the following example we initialize many times this code (from different terminals, where the nodes of our network are represented by individual processes)

```
leader_election.py
1 import etcd3
2
3 import sys
4 import time
5 from threading import Event
6
7
8 # The current leader is going to be the value with this key.
9 LEADER_KEY = "/algoexpert/leader"
10
11
12 # Entrypoint of the program.
13 def main(server_name):
14     # Create a new client to etcd.
15     client = etcd3.client(host="localhost", port=2379)
16
17     while True:
18         is_leader, lease = leader_election(client, server_name)
19
20         if is_leader:
21             print("I am the leader.")
22             on_leadership_gained(lease)
23         else:
24             print("I am a follower.")
25             wait_for_next_election(client)
26
def on_leadership_gained(lease):
    while True:
        # As long as this process is alive and we're the leader,
        # we try to renew the lease. We don't give up leadership
        # unless the process / machine crashes or some exception
        # is raised.
        try:
            print("Refreshing lease; still the leader.")
            lease.refresh()
            # This is where the business logic would go.
            do_work()
        except Exception:
            # Here we most likely got a client timeout (from
            # network issue). Try to revoke the current lease
            # so another member can get leadership.
            lease.revoke()
            return
        except KeyboardInterrupt:
            print("\nRevoking lease; no longer the leader.")
            # Here we're killing the process. Revoke the lease and exit.
            lease.revoke()
            sys.exit(1)
```

```

28 # This election mechanism consists of all clients trying to put their name
29 # into a single key, but in a way that only works if the key does not
30 # exist (or has expired before).
31 def leader_election(client, server_name):
32     print("New leader election happening.")
33     # Create a lease before creating a key. This way, if this client ever
34     # lets the lease expire, the keys associated with that lease will all
35     # expire as well.
36     # Here, if the client fails to renew lease for 5 seconds (network
37     # partition or machine goes down), then the leader election key will
38     # expire.
39     # https://help.compose.com/docs/etcd-using-etcd3-features#section-leases
40     lease = client.lease(5)
41
42     # Try to create the key with your name as the value. If it fails, then
43     # another server got there first.
44     is_leader = try_insert(client, LEADER_KEY, server_name, lease)
45     return is_leader, lease

```

Client.lease(5) the leader has to refresh the lease within 5 secs otherwise it expires, this key is deleted and a new leader can be set.

In the wait for next election we essentially watching for a Delete event on the leader key.

If something happens to the leader (shuts down for any reason or maybe its work lasts long so that it can't refresh its lease on time) someone of the followers will become the leader (the one who will write the key first).

```

def wait_for_next_election(client):
    election_event = Event()

    def watch_callback(resp):
        for event in resp.events:
            # For each event in the watch event, if the event is a deletion
            # it means the key expired / got deleted, which means the
            # leadership is up for grabs.
            if isinstance(event, etcd3.events.DeleteEvent):
                print("LEADERSHIP CHANGE REQUIRED")
                election_event.set()

    watch_id = client.add_watch_callback(LEADER_KEY, watch_callback)

    # While we haven't seen that leadership needs change, just sleep.
    try:
        while not election_event.is_set():
            time.sleep(1)
    except KeyboardInterrupt:
        client.cancel_watch(watch_id)
        sys.exit(1)

    # Cancel the watch; we see that election should happen again.
    client.cancel_watch(watch_id)

```

```

# Try to insert a key into etcd with a value and a lease. If the lease expires
# that key will get automatically deleted behind the scenes. If that key
# was already present, this will raise an exception.
def try_insert(client, key, value, lease):
    insert_succeeded, _ = client.transaction(
        failure=[],
        success=[client.transactions.put(key, value, lease)],
        compare=[client.transactions.version(key) == 0],
    )
    return insert_succeeded

def do_work():
    time.sleep(1)

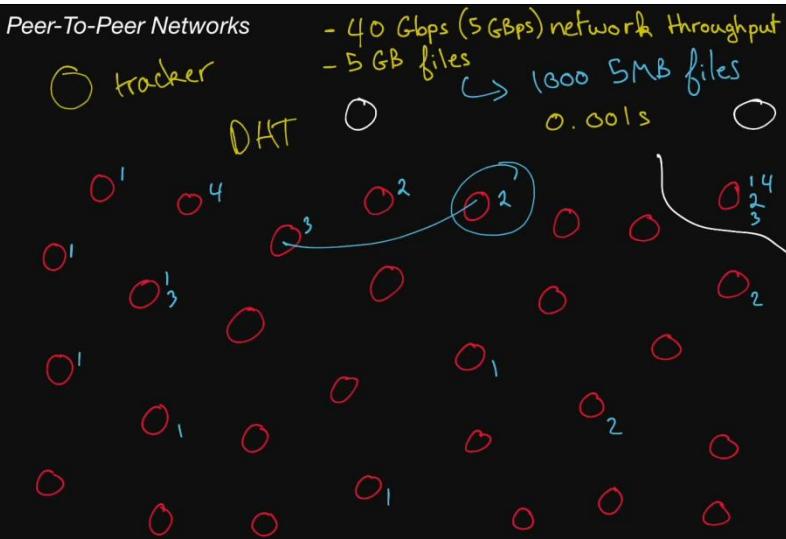
if __name__ == "__main__":
    server_name = sys.argv[1]
    main(server_name)

```

Strong consistency

Any node of a group, is sure that it will read the latest value from a the key value store.

Peer to Peer networks



Imagine that we have a network of 1000 machines to which we need to deploy an update, or transmit a large file to. Suppose that the file is 5 GB in size and our network throughput is 5 GBps (very fast, data center like fast). Initially the file exists in one machine.

In a non peer to peer architecture you would need to transfer the file to each machine individually. This task would require 1000 seconds (17 minutes) because 5 GB can be transferred in 1 second in a 5 GBps network.

In a peer to peer architecture instead, the large file is split to small chunks, let's say 1000 chunks of 5 MB each. These chunks are numbered so that the nodes can know how to recombine them. A 5 MB chunk can be transmitted in 1/1000 of a second. This is the time step. In the first time step 1 chunk is transmitted to one peer. In the second timestep, another chunk (chunk 2) is transmitted to another peer, and in addition the previously transmitted chunk 1 is transmitted by the peer that has it, to another

peer. So in the second timestep there have been transferred 2 chunks, or 100 MB. In the next step even more chunks are transmitted between the peers. This makes the whole process much much faster (600x faster)! The fundamental reason is that the peers can exchange data with each other, so in one timestep we have multiple transfers. All the peers have the complete file within a few seconds. 1000 chunks need 1 sec (initial node to peers) and at the same second most peers would have the complete file, taken from the other peers.

In the process, each peer should have some basic knowledge about what peers to communicate with next, what peers to transfer data to next. There are two ways to handle this. The first one is to have an orchestrator called tracker, that keeps this information and with which all peers communicate. The second one, is a gossip protocol (or epidemic protocol alternatively). In gossiping, each peer has a hash table which maps chunk numbers to peer addresses. The table is gradually filled. When a peer communicates with another peer, it sees what chunks this peer has in order to get some if it needs, and it also gets its hash table to see which peers has it communicated with previously and what chunks they had. This way it would know to which peer to communicate next. The individual hash tables of the peers, form a distributed hash table (DST) that maps chunks to peers.

Publish Subscribe

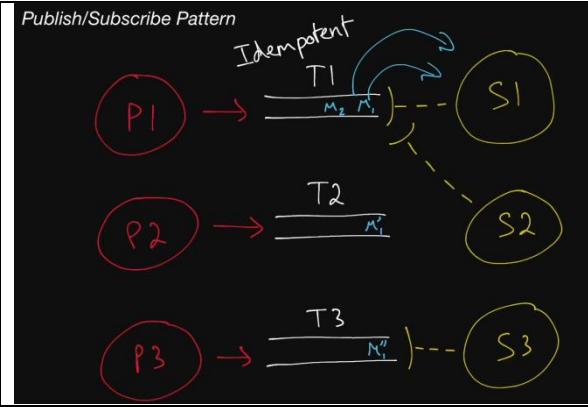
Just for clarification, this is different from the publish subscribe design pattern which deals with the relationship between objects of your application. This is a system design pattern that deals with the relationship between various subsystems of our system.

Messaging has two models: queuing and publish-subscribe

In queuing only the first available worker will execute the job (will get the message which is a job in this case and execute it), while in the publish subscribe model all subscribers would have to receive the message and execute it. have in mind that ordering is not guaranteed in queuing. There is order in the broker, but if you have many workers, they will get the messages in order, but there is no guarantee that they will execute it in order. One of them might delay for whatever reason etc.

This pattern can be useful for things the publishers need to execute asynchronously. It can also be a solution for the issue of live updates. Instead of having the clients connected to the servers directly, which can cause issues if the servers go down or if you have a network partition, the clients are subscribed (via websockets) to topics, to which the publishers push. Think of the topics as services with persistent storage.

The first class of cases that comes in mind is the one in which the publishers are back end services while the clients are the subscribers. For example your service publishes stock and crypto prices and the clients are subscribed to topic services which are also part of your back end system. Are there cases where the Publishers are clients and our back end implements the topic and subscriber services? Notice that in this case all subscribers would execute the message, it's not the typical queue with workers case. For example in a crypto exchange, the buy/sell orders can be the messages that go to a specific topic but only one subscriber should execute it.



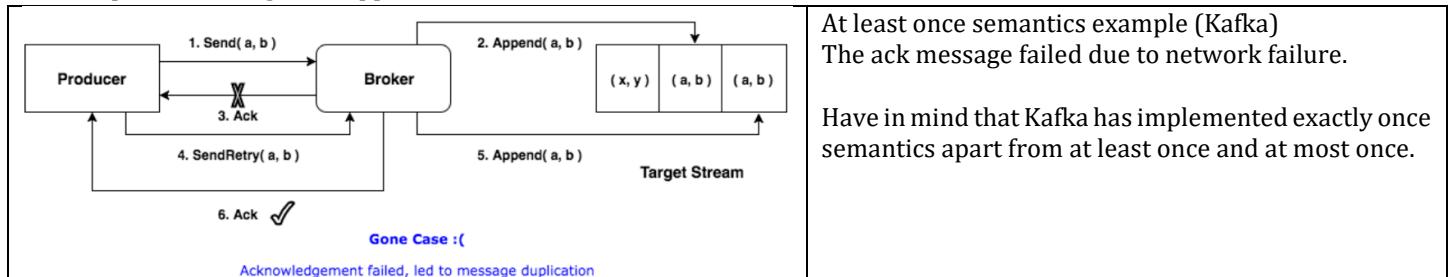
This pattern has 4 basic components. Publishers, subscribers, topics and messages. Subscribers are subscribed to topics (channels). Publishers push data to topics. The messages are not to be considered chat messages. They are generic. They can be operations for example that execute a specific logic. Usually you have more than one topics. You can have topic per data type, per region or per whatever else. For example you can publish stock prices to one topic and crypto prices to another.

A nice feature of pub/sub systems is that you can apply content filtering on the subscribers. For example one subscriber to the stock prices might filter out any non tech companies.

Common pub/sub services are [Apache Kafka](#) and [Google Cloud Pub/Sub](#). Google's solution offers autoscaling. This means for example that you don't have to externally apply sharding to topics when the data is large. Autoscaling in the topics level is provided by default. They also offer end to end encryption so only the subscribers know how to decode a message.

Usually the topics offer persistent storage so that the messages aren't lost in case of down time.

- Persistence usually offers at least once delivery, which means that when a topic sends a message it waits for an acknowledgement from the subscriber that received it. If the subscriber receives the message but breaks before it sends an acknowledgement then the publisher will send the message again. This means that in at least once delivery systems the messages must represent idempotent operations so that there will be no side effects even if the message is executed more than once. An idempotent operation has the same outcome no matter how many times it has been performed. A typical example of an idempotent operation is to have a completed status on the operation. So the next time, if it is already completed nothing will happen.



- Persistence also offers the possibility to apply message ordering. Messages are delivered to subscribers in the order they were published, like in a queue system.
- Persistence also offers replayability. You can replay past messages.
- You can also have end to end encryption so only the subscribers can decode the messages.

Messaging semantics

(Despite the fact that this is described within the Kafka context, it is a universal thing). In a distributed publish-subscribe messaging system, the computers that make up the system can always fail independently of one another. In the case of Kafka, an individual broker can crash, or a network failure can happen while the producer is sending a message to a topic. Depending on the action the producer takes to handle such a failure, you can get different semantics:

1. At-least-once semantics: if the producer receives an acknowledgement (ack) from the Kafka broker and acks=all, it means that the message has been written exactly once to the Kafka topic. However, if a producer ack times out or receives an error, it might retry sending the message assuming that the message was not written to the Kafka topic. If the broker had failed right before it sent the ack but after the message was successfully written to the Kafka topic, this retry leads to the message being written twice and hence delivered more than once to the end consumer. And everybody loves a cheerful giver, but

this approach can lead to duplicated work and incorrect results. So consumers should process messages in an idempotent way. This way a message could be processed more than one times without side effects.

2. **At-most-once semantics:** if the producer does not retry when an ack times out or returns an error, then the message might end up not being written to the Kafka topic, and hence not delivered to the consumer. In most cases it will be, but in order to avoid the possibility of duplication, we accept that sometimes messages will not get through.
3. **Exactly-once semantics:** even if a producer retries sending a message, it leads to the message being delivered exactly once to the end consumer. Exactly-once semantics is the most desirable guarantee, but also a poorly understood one. This is because it requires a cooperation between the messaging system itself and the application producing and consuming the messages. For instance, if after consuming a message successfully you rewind your Kafka consumer to a previous offset, you will receive all the messages from that offset to the latest one, all over again. This shows why the messaging system and the client application must cooperate to make exactly-once semantics happen.

Exactly-once delivery is one of the hardest problems to solve in distributed systems.

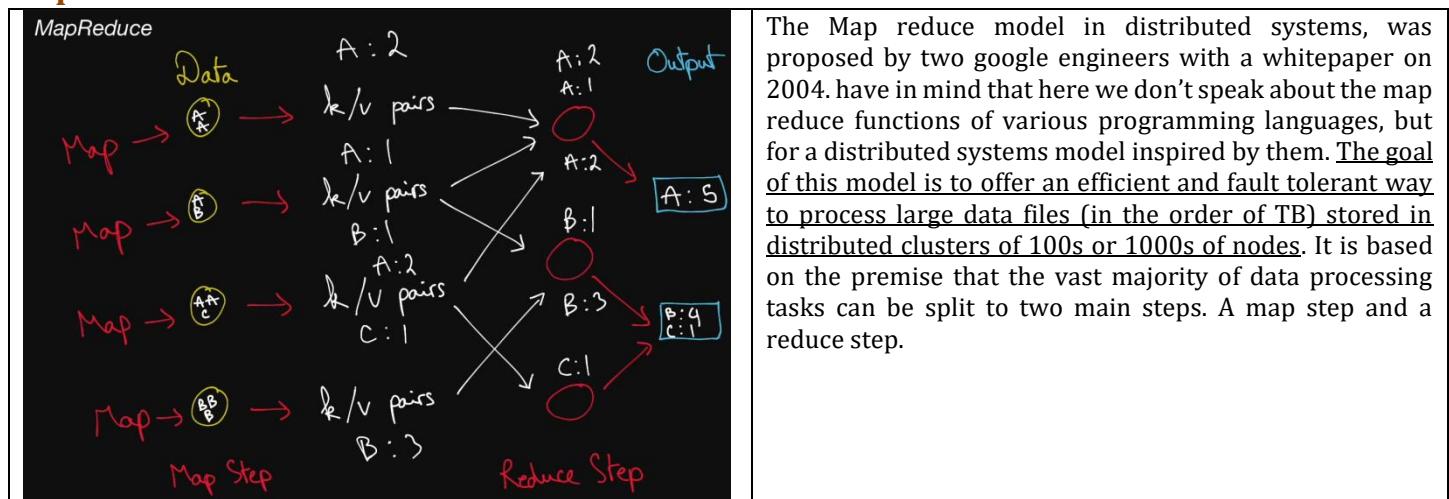
Async Tasks

In Two Scoops Of Django 3.x, in the chapter reserved for task Queues the authors say:

If time permits, move all asynchronous processes to Serverless systems such as AWS Lambda.

You can actually catch the result of our lambda after calling it, if you want it to be synchronous. If it's a longer running task then you could certainly do a post request at the end of the lambda call to write back instead. A great thing lambda does for you is offload RAM requirements for heavier workloads, so your celery process will no longer bog you down if it's a bottleneck for your service. Be warned, lambdas have a timeout of 15min max, so anything longer than that should be in an ECS task. The AWS free tier also allows 1m lambda requests per month, so it's a great option for minimizing your web instance footprint if you do have compute intensive tasks you'd like to offload.

Map Reduce



The data is split in many chunks and is spread across the whole cluster so that each node contains a piece of the whole data set. The map reduce model has been implemented as a framework that offers a distributed file system (DFS) which acts as a central plane above the cluster, is aware of all the nodes and the data set and knows how to orchestrate them. It knows how to communicate with nodes, with map workers, with reduce workers, where the output files live etc.

A typical example of a map reduce application, is to find the occurrence of all words within a large text file which is stored in a cluster of nodes. In the map step a specific function, the map function is applied on the data of each individual node. The input to this map function is the data of the node and the output is a set of key value pairs. For example the key could be a specific

word and the value could be the number of occurrences of that word in the text of this individual node. (have in mind that instead of such a key value output, the map function can emit key value pairs as it runs. For example it would emit a word as key and value 1 as value for each word). Important: to make the system fault tolerant, the map operations should be idempotent. So when a node fails before finishing the map step, the map step would be repeated and the outcome will not change. Have also in mind that the data stored in the nodes should not be moved because it is very large and, the map operations run locally to where the data lives.

After map there is an intermediate step between map and reduce. A shuffle step where the key value pairs are reorganized in a way. For example we might move all identical keys of each node, in one specific node in order to access this key more easily without having to access all nodes.

Then there is the reduce step which is applied on the shuffled data of each individual node and aggregates it to a specific value. For example it adds all the values of an identical key to calculate the total number of occurrence of a word in the whole data set.

There are various map reduce implementations which we can use. We don't have to create this whole system ourselves. Our job as administrators of that system, is to specify the map function, the reduce function and their input and output.

Another application is to have your log data stored in many nodes and you want to compute number of events per service or for a specific time frame.

The Rest

Live client updates

If you have an application that needs something like live updates for example a chat app, or odds in live betting, or bids and asks in an exchange, etc. You can implement this either with the typical request response model using polling, or with the event driven communication model using websockets. Your choice depends on the specific application. In many cases polling is enough, but not for chat apps where to provide a nice experience you would have very fast polling which can overload your servers with many requests. The generic term that algoexpert uses to describe this choice is polling vs streaming.

Configuration

All the configuration settings for your application live in a separate file in json or yml format. There are two kinds of configuration, static and dynamic. In static configuration the config file is part of your application package which means that if you make a change on it, you have to redeploy your whole application in order to see the effect. This has certain advantages like the fact that your new deployment has to be code reviewed and pass through tests before being deployed. But it makes configuration changes a bit slow and difficult. In dynamic configuration, the config file is not part of your application package. It is stored in a separate database that your application can query to get the configuration. This means that any change in the config file will be immediately available in your application without a need for new deployment. But you lack code review and passing the tests. You can build ui around your config file, so that you can change your settings through a gui. You can put access control on it (only certain users can modify settings).

Rate limiting

You can apply rate limiting per user, ip, region or anything else you want. You can have a distinct service that contains the rate limiting database (redis for example) that contains for example *user: last_run_timestamp*. Then your application servers would first communicate with redis to get the value and apply the rate limit logic to approve or not the request. Any service of your system can have its own rate limiting logic within it.

Have in mind tiers based rate limiting. For example you can allow users to perform a certain operation not more than 0.5 seconds apart, but then limit the amount of requests within one minute, and then the amount of requests within 1 hour etc. Have in mind that with ip based rate limiting you can avoid DoS attacks but not DDoS attacks that use different ip per request.

There are various rate limiting strategies. For example “pass through strategy” where a service which forwards a request to another service doesn’t apply rate limiting itself but relies on the other one to do it. If you are running an API service that connects to a legacy backend system that is not resilient under heavy loads, the API service should not use the pass-through strategy assuming that the legacy service will provide its own rate-limiting signals.

Debounce

Have also this term in mind. it refers to setting a timer in the client within which you don’t allow them to perform an action. You set a debounce.

Logging and monitoring

Logging is extremely important. There are libraries that create a nice format for the messages. Log messages are written in syslog or json format usually.

Monitoring. It is very important to have systems in place to gather metrics and then monitor them (and create alerts too). There are usually two types of monitoring tools. One type, scraps the logging content to extract metrics and allow you to monitor them. But this requires that you log all the metrics you care about and this might not be convenient. Also if the format of logging changes you should update the scrapping. Another way is to store all the metrics that you care about in a time series database and use a reporting tool on top of that database. There are tools that create graphs out of them (graphana) or you can query this database. You can also integrate messaging apps to the reporting tool to get alerts on important events.

Communications security

Man in the middle attack (MITM attack) is a very common attack in http, where a third party can intercept the communication and read or alter it. For this reason http needs encryption.

There are two kinds of encryption. Symmetric where only one key is used to encrypt and decrypt. This means that the two parties have to share this key between them which makes it vulnerable to MITM attack. AES (advanced encryption standard) is a symmetric encryption standard. Assymetric encryption uses two keys which are generated together in such a way, that whatever is encrypted with one key of the pair, can be decrypted with the other.

HTTPS is HTTP over TLS. TLS is a mechanism for securing a connection using asymmetric encryption. The key used for encryption is called the public key because it is shared between the communication parties and the one used for decryption is called the private key.

A secure connection is established with a TLS handshake.

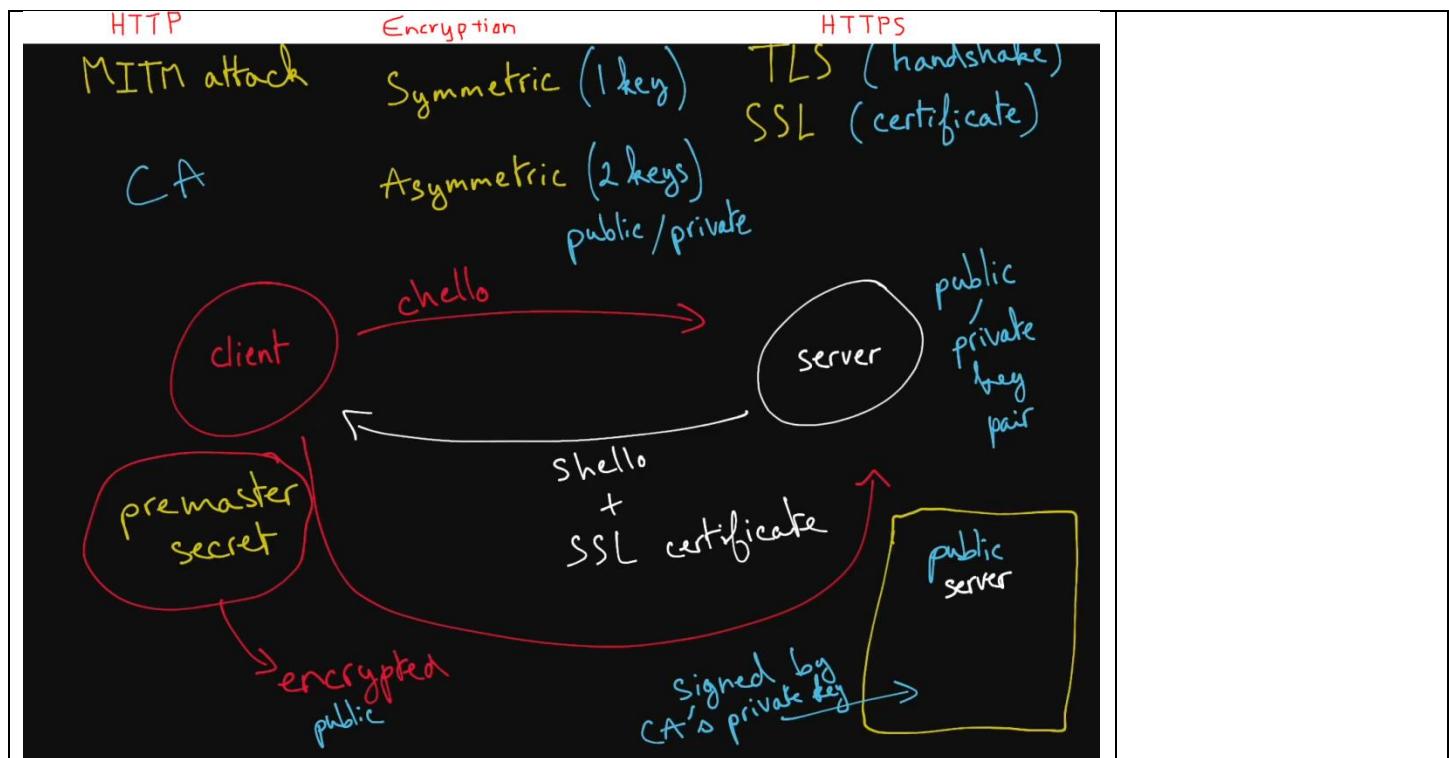
1. The client sends a hello message to the server. It is a string of bytes.
2. The server responds with a hello message which is another string of bytes. It also sends an SSL certificate along with the hello message. The ssl certificate contains the public key of the server.

3. The client generates another message, the premaster secret as it is called, encrypts it with the public key and sends it to the server.

4. The server and the client use the client hello (chello), the server hello (shello) and the premaster secret to generate a set of 4 keys that are specific to this session and that can be thought of as a single symmetric key that both the server and the client can use to encrypt and decrypt their messages (for as long as this session lasts). both parties can generate these keys independently I guess.

5. There is one additional message exchange in the handshake, encrypted with the symmetric key, which says that the handshake has ended successfully. From now on the connection is secure.

Notice that in this setting a 3rd party can intercept the shello + SSL certificate message which is unencrypted, change the public key with its own key and then the client would encrypt with this public key which means that the attacker will be able to decrypt the messages. The solution is SSL certificates issued and cryptographically signed by a trusted 3rd party, a certificate authority (CA). the server communicates with the CA, which is the one that generates the key pair for the server, ensures that the server is who it says it is and creates a certificate for this, which it signs with the authority's private key. So the SSL certificate is signed (encrypted) and the client has to get the authority's public key to decrypt it. This way no one can intercept and alter the SSL certificate. Usually browsers store the public keys of known CAs.



REST API design

<https://www.youtube.com/watch?v=P0a7PwRNLVU> design quality APIs

API design is one type of interview in which you are asked to design an API, for example design twitter api. You have to make questions about which specific part of twitter you have to design etc. After that you have to start thinking about

- What resources you need to have
- What attributes these resources would have
- What endpoints per resource and what parameters, some of them required.

Some notes

✓ **Pagination.**

For example with a limit and a starting_after or ending_before parameters (as stripe does). you use limit with one of the other two which receives an id. The api orders the objects in reverse chronological order and returns the specified objects (after the given id or before it)

Or a limit and a next parameter. The next parameter has the role of starting_after.

Or you could have a page parameter where each page has a standard number of objects (or you can have a per_page parameter too).

All these parameters are passed as url query string parameters.

✓ **Sorting**

You can have a sort field. Let the sort parameter take in list of comma separated fields *GET /tickets?sort=-priority*

✓ **Filtering** (Selecting which attributes you want)

You can have a fields (or fieldmask) query string parameter (comma separated list for example) in some large resources with many attributes in order to specify what attributes of the resource you want in the response.

GET /tickets?fields=id,subject,customer_name,updated_at&state=open&sort=-updated_at

✓ **Relations**

How will you handle related fields, will you include the whole object or a reference to it? You can return ids by default, but you can return an object with specific attributes if the client requests them with an “include” (or embed or expand) like url query string parameter, like in sportmonks. For example:

https://soccer.sportmonks.com/api/v2.0/fixtures/between/{start_date}/{end_date}?include=seasons.name,home_team.name

The fixture object has references to the season and home_team objects. You can specify this way what attributes of the references you want to receive in the response.

```
GET /tickets/12?embed=customer.name,assigned_user
{
  "id" : 12,
  "subject" : "I have a question!",
  "summary" : "Hi, ....",
  "customer" : {
    "name" : "Bob"
  },
  assigned_user: {
    "id" : 42,
    "name" : "Jim",
  }
}
```

There would be a significant efficiency gain from allowing related data to be returned and loaded alongside the original resource on demand. However, as this does go against some RESTful principles, we can minimize our deviation by only doing so based on an embed (or expand, or include) query parameter. In this case, embed would be a comma separated list of fields to be embedded. Dot-notation could be used to refer to sub-fields.

I prefer the second approach, because it doesn't add the internals of our app to the endpoint that represents a resource. Our internal logic, goes to the queryset.

<https://api.zakanda.com/v1/totalbets/1/betevents>

<https://api.zakanda.com/v1/betevents?totalbet=1>

✓ **Searching**

Stripe uses a query language as they call it. you put the query in a “query” parameter the value of which is url encoded by your client library.

[https://api.stripe.com/v1/charges/search?query=amount>999 AND metadata\['order_id'\]:6735"](https://api.stripe.com/v1/charges/search?query=amount>999 AND metadata['order_id']:6735)

they have some rules for the query body.

✓ **Endpoints for Websockets**

You can have endpoints that open websocket connections with the server. For example in a chat app you can have a **StreamMessages** endpoint (“endpoint with Stream prefix instead of “Get” like GetMessages) that opens a websocket connection with the server and receives the chat messages.

- ✓ **Versioning.** Have the version in the url instead of as a header.
- ✓ Authentication token should be specified in the basic authentication header (for example in the username header) or as a bearer token (Authorization: Bearer your_token).
- ✓ Get requests can send data (parameter values) through the query string
- ✓ Post requests can send data in the http request body using various content types (json-formatted, form url encoded)
- ✓ Given entities we shouldn't deal with

The interviewer could tell us to not worry about the details of a specific action for which we can assume that we already have a given entity that represents that action. For example on a subscription service with payments he might tell us not to worry about the internals of subscription and payments (duration, amount, recurrent, payment method etc.) and just assume we have two entities the SubscriptionInfo and PaymentInfo which contains the relevant information and just use these. So we could define a CreateSubscription endpoint like this:

CreateSubscription(sellerId: str, subInfo: SubscriptionInfo, payInfo: PaymentInfo) => Subscription entity

Or another example of an endpoint that opens a websocket and returns a video stream. You don't have to deal with video hosting, encodings etc.) you are just told that a VideoInfo entity is returned by that endpoint, that contains a lot of data that you don't have to worry about.

StreamVideo instead of GetVideo (channelId: str) => VideoInfo entity

- ✓ Python's request library payload parameter adds the payload keys in the query string

Api specification

You can use a simple text file to write your design. you could use swagger if you want which is an IDL (interface description language), a format to describe apis. You can check various apis (stripe, youtube, twitter etc.) to see how they do it.

```

api.txt
1 # API Definition
2
3 ## Entity Definitions
4
5 class Charge:
6     - id: uuid
7     - customer_id: uuid
8     - amount: integer
9     - currency: string (or currency-code enum)
10    - status: enum ("succeeded", "pending", "failed")
11
12 class Customer:
13     - id: uuid
14     - name: string
15     - address: string
16     - email: string
17     - card: Card
18
19 class Card:
20
21 ## Endpoint Definitions
22 ## Charges
23 CreateCharge(charge: Charge)
24     => Charge
25 GetCharge(id: uuid)
26     => Charge
27 UpdateCharge(id: uuid, updatedCharge: Charge)
28     => Charge
29 ListCharges(offset: integer, limit: integer)
30     => Charge[]
31 CaptureCharge(id: uuid)
32     => Charge
33
34 ## Customers
35 CreateCustomer(customer: Customer)
36     => Customer
37 GetCustomer(id: uuid)
38     => Customer
39 UpdateCustomer(id: uuid, updatedCustomer: Customer)
40     => Customer
41 DeleteCustomer(id: uuid)
42     => Customer
43 ListCustomers(offset: integer, limit: integer)
44     => Customer[]

charge.json
1 {
2     "swagger": "2.0",
3     "info": {
4         "version": "1.0.0",
5         "title": "Example Stripe API"
6     },
7     "host": "api.stripe.com",
8     "basePath": "/v1",
9     "schemes": [ "http", "https" ],
10    "consumes": [ "application/json" ],
11    "produces": [ "application/json" ],
12    "paths": {
13        "/v1/charges": {
14            "get": {
15                "summary": "List all charges",
16                "operationId": "listCharges",
17                "parameters": [
18                    {
19                        "name": "offset",
20                        "in": "query",
21                        "description": "How many items to skip in the result set",
22                        "required": false,
23                        "type": "integer",
24                        "format": "int32"
25                    },
26                    {
27                        "name": "limit",
28                        "in": "query",
29                        "description": "How many items to return at once",
30                        "required": false,
31                        "type": "integer",
32                        "format": "int32"
33                    }
34                ],
35                "responses": {
36                    "200": {
37                        "description": "A paginated array of charges",
38                        "schema": {
39                            "$ref": "#/definitions/Charges"
40                        }
41                    }
42                }
43            }
44        }
45    }
46}

customer.json
1 {
2     "swagger": "2.0",
3     "info": {
4         "version": "1.0.0",
5         "title": "Example Stripe API"
6     },
7     "host": "api.stripe.com",
8     "basePath": "/v1",
9     "schemes": [ "http", "https" ],
10    "consumes": [ "application/json" ],
11    "produces": [ "application/json" ],
12    "paths": {
13        "/v1/customers": {
14            "get": {
15                "summary": "List all customers",
16                "operationId": "listCustomers",
17                "parameters": [
18                    {
19                        "name": "offset",
20                        "in": "query",
21                        "description": "How many items to skip in the result set",
22                        "required": false,
23                        "type": "integer",
24                        "format": "int32"
25                    },
26                    {
27                        "name": "limit",
28                        "in": "query",
29                        "description": "How many items to return at once",
30                        "required": false,
31                        "type": "integer",
32                        "format": "int32"
33                    }
34                ],
35                "responses": {
36                    "200": {
37                        "description": "A paginated array of customers",
38                        "schema": {
39                            "$ref": "#/definitions/Customer"
40                        }
41                    }
42                }
43            }
44        }
45    }
46}

```

Filtering, sorting & searching

Complex result filters, sorting requirements and advanced searching (when restricted to a single type of resource) can all be easily implemented as query parameters on top of the base URL.

Filtering: GET /tickets?state=open define the attributes you want

Sorting: Similar to filtering, a generic parameter `sort` can be used to describe sorting rules. Let the sort parameter take in list of comma separated fields GET /tickets?sort=-priority

Searching: Sometimes basic filters aren't enough and you need the power of full text search. Perhaps you're already using ElasticSearch or another Lucene based search technology. Use a query parameter (eg. q in this example)

- GET /tickets?sort=-updated_at - Retrieve recently updated tickets
- GET /tickets?state=closed&sort=-updated_at - Retrieve recently closed tickets
- GET /tickets?q=return&state=open&sort=-priority,created_at - Retrieve the highest priority open tickets mentioning the word 'return'

To make the API experience more pleasant for the average consumer, consider packaging up sets of conditions into easily accessible RESTful paths. Eg. the recently closed tickets query above could be packaged up as GET /tickets/recently_closed

Pagination

Envelope loving APIs typically include pagination data in the envelope itself. And I don't blame them - until recently, there weren't many better options. The right way to include pagination details today is using the Link header introduced by RFC 5988. The **Link entity-header** field provides a means for serialising one or more links in HTTP headers. It is semantically equivalent to the <LINK> element in HTML. An example from github api (this is in response's header)

Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next", <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"

Rate limiting

At a minimum, include the following headers (using Twitter's naming conventions as headers typically don't have mid-word capitalization):

- X-Rate-Limit-Limit - The number of allowed requests in the current period
- X-Rate-Limit-Remaining - The number of remaining requests in the current period
- X-Rate-Limit-Reset - The number of seconds left in the current period

Caching

HTTP provides a built-in caching framework! All you have to do is include some additional outbound response headers and do a little validation when you receive some inbound request headers.

Headers: ETag and Last-Modified

Errors

```
{
  "code" : 1024,
  "message" : "Validation Failed",
  "errors" : [
    {
      "code" : 5432,
      "field" : "first_name",
      "message" : "First name cannot have fancy characters"
    },
    {
      "code" : 5622,
      "field" : "password",
      "message" : "Password cannot be blank"
    }
  ]
}
```

Just like an HTML error page shows a useful error message to a visitor, an API should provide a useful error message in a known consumable format.

HTTP status codes

your api should at least use these

- 200 OK - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that doesn't result in a creation.

- 201 Created - Response to a POST that results in a creation. Should be combined with a [Location header](#) pointing to the location of the new resource
- 204 No Content - Response to a successful request that won't be returning a body (like a DELETE request)
- 304 Not Modified - Used when HTTP caching headers are in play
- 400 Bad Request - The request is malformed, such as if the body does not parse
- 401 Unauthorized - When no or invalid authentication details are provided. Also useful to trigger an auth popup if the API is used from a browser
- 403 Forbidden - When authentication succeeded but authenticated user doesn't have access to the resource
- 404 Not Found - When a non-existent resource is requested
- 405 Method Not Allowed - When an HTTP method is being requested that isn't allowed for the authenticated user
- 410 Gone - Indicates that the resource at this end point is no longer available. Useful as a blanket response for old API versions
- 415 Unsupported Media Type - If incorrect content type was provided as part of the request
- 422 Unprocessable Entity - Used for validation errors
- 429 Too Many Requests - When a request is rejected due to rate limiting

Typical First Mistake	
Typical	Better
{ "id": "12345", "type": "/dog", "name": "Lassie", "furColor": "brown" }	{ "id": "/pet/12345", "type": "/dog", "name": "Lassie", "furColor": "brown" }
Typical approach defeats the "uniform API". Client now needs application-specific knowledge to convert "12345" to "/pet/12345"	

Relationships		Relationships between entities of different applications
Typical	Better	If the owner is in another system you write the whole url. If it is in the same system, you can write only the relative part.
{ "id": "12345", "type": "/dog", "name": "Lassie", "furColor": "brown", "ownerID": "98765" }	{ "id": "/pet/12345", "type": "/dog", "name": "Lassie", "furColor": "brown", "owner": "https://tracker.com/owner/98765" }	

Swagger (now OpenAPI)

It is a standard to document APIs using json.

There are web interfaces that read the api specification (the swagger schema) and automatically create documentation for it.

At some point, Swagger was given to the Linux Foundation, to be renamed OpenAPI. That's why when talking about version 2.0 it's common to say "Swagger", and for version 3+ "OpenAPI"

Schema

```

1. {
2.   "type": "object",
3.   "required": [
4.     "name"
5.   ],
6.   "properties": {
7.     "name": {
8.       "type": "string"
9.     },
10.    "address": {
11.      "$ref": "#/components/schemas/Address"
12.    },
13.    "age": {
14.      "type": "integer",
15.      "format": "int32",
16.      "minimum": 0
17.    }
18.  }
19. }

```

```

1. type: object
2. required:
3. - name
4. properties:
5.   name:
6.     type: string
7.   address:
8.     $ref: '#/components/schemas/Address'
9.   age:
10.    type: integer
11.    format: int32
12.    minimum: 0

```

API wrappers

The concept is that you create a package that reads the API endpoints and returns the data in language specific data structures (lists, dictionaries, class instances etc.) in order to be able to use them in an idiomatic way. The package must contain tests that verify the responses and how they are read. At the end you can configure a continuous automatic testing solution (Travis CI, semaphore CI etc.) that runs the tests regularly and inform you in cases of failure.

APIs are written as a rule to be for general use. They allow access to specific common functions. Wrappers make using the API easier.

- A wrapper can allow you to combine multiple calls to those functions, or help in making your particular program(s) do more complicated tasks. For instance the api may allow you to do a search, in the case of reddit on a specific subreddit. But in the program you are creating you may want to have a specifically targeted set of subreddits. Rather than call the api 3 times with your program, you can call the wrapper search function once which knows that it calls the api search function 3 times. The wrapper could allow you to search /askscience /theydidthemath /homeworkhelp all at the same time, whereas the api would require you to specify each subreddit individually.
- A wrapper can also be used to enforce certain controls that the api would otherwise allow. For instance when talking about an API that allows you to enter some text into a field, it would in most cases allow any text so that the API can be used in as many instances as possible. However, you know that the specific application you are using will never allow the % symbol. Your wrapper will validate that any text entered does not have a % in it before you tell the API to do anything with the text.

The concept

<https://semaphoreci.com/community/tutorials/building-and-testing-an-api-wrapper-in-python>

you have an api wrapper package for each api

- api1 package
It contains modules for each api endpoint group, like events.py to get list of events, info for an event etc.. ,The module has a Class with methods. Each method makes an api call and returns the data. The package contains tests so you can easily test if the api responses remain the same (if they are read as expected). The api key is stored as env variable and is used as a requests session parameter. The api responses can be saved in the disk so that the api isn't called every time you run a test. This can be done with the vcr package.
- api2 package

in your code:

```
from api1 import Event  
  
# from api2.events import Event  
  
Event.get_events()
```

The get_events is an Event method that does the API call and returns them as python data structure. So you could change the api1 to api2 using exactly the same code except from importing api2.

Testing the API response

If you want to be sure that the api response contains the data attributes that you expect you must check this with the response. In an example that I saw he did it using pytest.fixture to implement this in tests but the concept is the same. You create a list of the expected attributes. Then you make it a set and check if it is a subset of the response keys (for which you have created a set)

# tests/test_tmdbwrapper.py	testing the API response
<pre>@vcr.use_cassette('tests/vcr_cassettes/tv-popular.yml') def test_tv_popular(): """Tests an API call to get a popular tv shows""" response = TV.popular() assert isinstance(response, dict) assert isinstance(response['results'], list) assert isinstance(response['results'][0], dict) assert set(tv_keys).issubset(response['results'][0].keys())</pre>	

requests.session

Within the __init__ of the api1 package. So every time you import the package the session is created and contains the key. Then in the code you call api endpoints using the session variable.

```
session = requests.Session()  
session.params = {}  
session.params['api_key'] = TMDB_API_KEY  
  
def info(self):  
    path = 'https://api.themoviedb.org/3/tv/{}'.format(self.id)  
    response = session.get(path)  
    return response.json()
```

Then, we go ahead and initialize a requests session and provide the API key in the params object. This means that it will be appended as a parameter to each request we make with this session object.

So when you want to make a call to an API endpoint, you use the session that you have already created. Each API response needs its own file.

The session definition could be done in the __init__ file of the package so that you can use it directly within the package.

vcr package

A better way would be to save the HTTP response the first time a request is made, then reuse this saved response on subsequent test runs. This way, we minimize the amount of requests we need to make on the API and ensure that our tests still have access to the correct data. To accomplish this, we will use the vcr package

<pre># tests/test_tmdbwrapper.py import vcr @vcr.use_cassette('tests/vcr_cassettes/tv-info.yml') def test_tv_info(tv_keys): """Tests an API call to get a TV show's info""" tv_instance = TV(1396) response = tv_instance.info() assert isinstance(response, dict) assert response['id'] == 1396, "The ID should be in the response" assert set(tv_keys).issubset(response.keys()), "All keys should be in the response"</pre>	<p>We just need to instruct vcr where to store the HTTP response for the request that will be made for any specific test.</p> <p>Then each time you run this function that makes a call to the api (response = tv_instance.info()), no actual api call will be made. The already saved data will be returned instead.</p>
<pre>@vcr.use_cassette('tests/vcr_cassettes/tv-popular.yml', filter_query_parameters=['api_key'])</pre>	<p>Api response contains also the API Key of the request. Since vcr stores the responses in the disk its a good idea to remove the api key from the response so that it is not stored in the disk, for security reasons.</p> <p>The filter_query_parameters arg makes that This will save the API responses, but it will leave out the API key.</p>

Checking if a list of user defined keys are indeed keys of a dictionary

Comes down to checking if a set is a subset of another one

Tv_keys is a list

```
set(tv_keys).issubset(response.keys())
```

Wrapper classes

A Wrapper Class (or the Wrapper Pattern) is where you declare a Class as a container for an Object to extend the functionality only for display or processing purposes (i.e. you don't intend for that attribute to be persisted)

Vcrpy

VCR records everything that is communicated over HTTP.

REST old

RPC style web APIs vs REST Style web APIs

<https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/>

You can implement custom actions into a REST api, but it would be difficult to respect its restfulness.

One simple rule of thumb is this:

- If an API is mostly actions, maybe it should be RPC.
- If an API is mostly CRUD and is manipulating related data, maybe it should be REST.

Or an RPC API might be a new service to complement an existing REST API. It's best not to mix styles in a single API, because this could be confusing both to consumers of your API as well as to any tools that expect one set of conventions (REST, for example) and that fall over when it instead sees a different set of conventions (RPC).

For example

"We have a REST API to manage a web hosting company. We can create new server instances and assign them to users, which works nicely, but how do we restart servers and run commands on batches of servers via the API in a RESTful way?"

There's no real way to do this that isn't horrible, other than creating a simple RPC-style service that has a POST /restartServer method and a POST /execServer method, which could be executed on servers built and maintained via the REST server.

<http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

Restful urls and actions

The key principles of REST involve separating your API into logical resources. These resources are manipulated using HTTP requests where the method (GET, POST, PUT, PATCH, DELETE) has specific meaning. But what can I make a resource? Well, these should be **nouns (not verbs!)** that make sense from the perspective of the API consumer. Although your internal models may map neatly to resources, it isn't necessarily a one-to-one mapping. The key here is to not leak irrelevant implementation details out to your API!

Very bad practice to use verbs in as api resources.		With just 2 urls we have plenty of actions				
Resource	POST create	GET read	PUT update	DELETE delete		
... /getAllDogs /verifyLocation /feedNeeded /createRecurringWakeUp /giveDirectOrder /checkHealth /getRecurringWakeUpSchedule /getLocation /getDog /newDog /getNewDogsSince /getRedDogs /getSittingDogs /getDogStateTo /replaceSittingDogsWithRunningDogs /saveDog /getAllLeashedDogs /verifyVeterinarianLocation /feedNeededFood /createRecurringMedication /doDirectOwnerDiscipline /doExpressCheckupWithVeterinarian /getRecurringFeedingSchedule /getHungerLevel /getSquirrelsChasingPuppies /newDogForOwner /getNewDogsAtKennelSince /getRedDogsWithoutSiblings /getSittingDogsAtPark /setLeashedDogStateTo /replaceParkSittingDogsWithRunningDogs /saveMommaDogsPuppies ...	/dogs	create a new dog	list dogs	bulk update dogs	delete all dogs
		/dogs/1234	error	show Bo	if exists update Bo if not error	delete Bo

Once you have your resources defined (eg. tickets, user, group etc), you need to identify what actions apply to them and how those would map to your API. RESTful principles provide strategies to handle CRUD actions using HTTP methods mapped as follows:

- GET /tickets - Retrieves a list of tickets
- GET /tickets/12 - Retrieves a specific ticket
- POST /tickets - Creates a new ticket
- PUT /tickets/12 - Updates ticket #12
- PATCH /tickets/12 - Partially updates ticket #12
- DELETE /tickets/12 - Deletes ticket #12

Here you have a one tickets/ endpoint.

Relations:

A ticket in Enchant consists of a number of messages. These messages can be logically mapped to the /tickets endpoint as follows:

- GET /tickets/12/messages - Retrieves list of messages for ticket #12
- GET /tickets/12/messages/5 - Retrieves message #5 for ticket #12

- POST /tickets/12/messages - Creates a new message in ticket #12
- PUT /tickets/12/messages/5 - Updates message #5 for ticket #12
- PATCH /tickets/12/messages/5 - Partially updates message #5 for ticket #12
- DELETE /tickets/12/messages/5 - Deletes message #5 for ticket #12

if the relation is commonly requested alongside the resource, the API could offer functionality to automatically embed the relation's representation and avoid the second hit to the API.

Always use SSL. No exceptions. One thing to watch out for is non-SSL access to API URLs. Do not redirect these to their SSL counterparts. Throw a hard error instead!

Versioning

There are mixed opinions around whether an API version should be included in the URL or in a header. Academically speaking, it should probably be in a header. However, the version needs to be in the URL to ensure browser explorability of the resources across versions (remember the API requirements specified at the top of this post?).

Updates & creation should return a resource representation

A PUT, POST or PATCH call may make modifications to fields of the underlying resource that weren't part of the provided parameters (for example: created_at or updated_at timestamps). To prevent an API consumer from having to hit the API again for an updated representation, have the API return the updated (or created) representation as part of the response.

In case of a POST that resulted in a creation, use a [HTTP 201 status code](#) and include a [Location header](#) that points to the URL of the new resource. (If a resource has been created on the origin server, the response SHOULD be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header)

Overriding the HTTP method

the popular convention is to accept a request header X-HTTP-Method-Override

Not use HATEOAS yet

There are a lot of mixed opinions as to whether the API consumer should create links or whether links should be provided by the API. RESTful design principles specify HATEOAS. Although the web generally works on HATEOAS type principles (where we go to a website's front page and follow links based on what we see on the page), I don't think we're ready for HATEOAS on APIs just yet. For now, it's best to assume the user has access to the documentation & the resource identifiers that the api must include in the output representation (in the response) which the API consumer will use when crafting links. Also, given this post advocates version numbers in the URL, it makes more sense in the long term for the API consumer to store resource identifiers as opposed to URLs. After all, the identifier is stable across versions but the URL representing it is not!

JSON

Your output representation's primary need is serialization from an internal representation (of data model, of the data stored in a db). If you have to support also xml, then should the media type change based on Accept headers or based on the URL? To ensure browser explorability, it should be in the URL. The most sensible option here would be to append a .json or .xml extension to the endpoint URL.

camelCase vs snake_case

conventions are snake_case for python & ruby, camelCase for javascript. Opinion: Use snake_case for the field names

Pretty Print by default & ensure gzip is supported

An API that provides white-space compressed output isn't very fun to look at from a browser. Although some sort of query parameter (like ?pretty=true) could be provided to enable pretty printing, an API that pretty prints by default is much more approachable. The cost of the extra data transfer is negligible, especially when you compare to the cost of not implementing gzip. GitHub's API, which uses pretty print by default

Don't use an envelope by default, but make it possible when needed

```
Many APIs wrap their
responses
in
envelopes like this:
{
  "data" : {
    "id" : 123,
    "name" : "John"
  }
}
```

There are a couple of justifications for doing this - it makes it easy to include additional metadata or pagination information, some REST clients don't allow easy access to HTTP headers & JSONP requests have no access to HTTP headers. However, with standards that are being rapidly adopted like CORS and the Link header from RFC 5988, enveloping is starting to become unnecessary. We can future proof the API by staying envelope free by default and enveloping only in exceptional cases.

There are 2 situations where an envelope is really needed

- if the API needs to support cross domain requests over JSONP or
- if the client is incapable of working with HTTP headers.

Json encoded POST, PUT, PATCH api request

This refers to JSON for API input (for API output we already saw that it's the recommended content type). It's **JSON vs URL encoding** for api's request body encoding. Many APIs use URL encoding in their API request bodies. URL encoding is exactly what it sounds like - request bodies where key value pairs are encoded using the same conventions as one would use to encode data in URL query parameters. URL encoding has a few issues that make it problematic. It has no concept of data types. This forces the API to parse integers and booleans out of strings. Furthermore, it has no real concept of hierarchical structure. If the API is simple, URL encoding may suffice. However, complex APIs should stick to JSON for their API input. Either way, pick one and be consistent throughout the API. An API that accepts JSON encoded POST, PUT & PATCH requests should also require the Content-Type header be set to application/json or throw a 415 Unsupported Media Type HTTP status code.

Do you have any specific reason to post JSON?

If not, then I would say that the default choice would be to post form-data, as that is natively supported by all browsers and all servers. You can post form-data without any client script at all, and any server implementation handles it by default.

Urlencoded can be more characters, so heavier

When submitting through JSON, you cannot send files? You can but you have to base64 encode them (basically encoding them to string) which will make them bigger than a binary but it is possible.

Sending data in JSON format is slightly more powerful than sending traditional POST data, because JSON allows you to send nested data.

```

POST /v1/send?access_token=YOUR_API_TOKEN HTTP/1.0
Host: www.smtpeter.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 1484

envelope=info%40example.com&recipient=....

```

```

POST /v1/send?access_token={YOUR_API_TOKEN} HTTP/1.0
Host: www.smtpeter.com
Content-Type: application/json
Content-Length: 246

{
    "envelope": "info@example.com",
    "recipient": "john@doe.com",
    "subject": "This is the subject",
    "html": "This is example content",
    "from": "info@example.com",
    "to": "john@doe.com"
}

```

Authentication

A RESTful API should be stateless. This means that request authentication should not depend on cookies or sessions. Instead, each request should come with some sort authentication credentials. All three methods described are just ways to transport the token across the API boundary.

- By always using SSL, the authentication credentials can be simplified to a randomly generated access token that is delivered in the user name field of HTTP Basic Auth. The great thing about this is that it's completely browser explorable - the browser will just popup a prompt asking for credentials if it receives a 401 Unauthorized status code from the server.
- However, this token-over-basic-auth method of authentication is only acceptable in cases where it's practical to have the user copy a token from an administration interface to the API consumer environment. In cases where this isn't possible, [OAuth 2](#) should be used to provide secure token transfer to a third party. OAuth 2 uses [Bearer tokens](#) & also depends on SSL for its underlying transport encryption.

Bearer token is a possible type (the default in most implementations) in the token_type parameter in oauth2. If you use this type, an access_token is generated and sent back to you. Bearer can be simply understood as "**give access to who ever brings the bearer token.**" One valid token and no question asked. Any party with a bearer token (a "bearer") can use it to get access to the associated oauth2 protected resources (without demonstrating possession of a cryptographic key). On the other hand if you choose Mac and sign_type (default hmac-sha-1 on most implementation), the access token is generated and kept as secret in Key Manager as a attribute, and an encrypted secret is sent back as access_token

Stripe uses basic auth or bearer token.

- An API that needs to support JSONP will need a third method of authentication, as JSONP requests cannot send HTTP Basic Auth credentials or Bearer tokens. In this case, a special query parameter `access_token` can be used. Note: there is an inherent security issue in using a query parameter for the token as most web servers store query parameters in server logs.

Some apis receive the key as a querystring parameter. This is not the preferred way I think (visible by all)

Maintaining state in REST

You shouldn't keep the state of a session in the server (like I do for the bet slip items for example) but in the client. A restfull service must be stateless. One reason is scalability issues: Let's try to add a load balancer and another service instance to your system. In this case you have to share the sessions between the service instances. It is hard to maintain and extend such a system, so it scales badly... On the other hand client side state does not violate REST principles. (**So you could use session cookies instead of session db table to maintain state and use token based authentication instead of session based which uses**

the db table. But cookies have a max available size almost 4MB so in case of large state data? For small data serialize them to json and store them to a cookie. The idea behind statelessness is that the SERVER is stateless, (not the clients). Any client issuing an identical request (same headers, cookies, URI, etc) should be taken to the same place in the application.

Notice that using a session store in your sql db or on a no-sql one like redis makes no difference as far as the state of the server is concerned. You store the state in the server.

RESTful services

Representational State Transfer

REST is an architectural style that describes the communication between a client and a server. The client can be a browser or a software. There are some common practices. A restful service uses 5 http methods.

- GET
- PUT (add something to a table) A PUT request is idempotent. Idempotency is the main difference between the expectations of PUT versus a POST request.
- HEAD (the response will be only the header not the body)
- OPTIONS (it will return what other methods the server supports)
- POST

Principles of REST

- **Resources** expose easily understood directory structure URLs.
- **Representations** transfer JSON or XML to represent data objects and attributes.
- **Messages** use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).
- **Stateless** interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.

So if the method is GET a restful service will act in the specified way and will only retrieve the data. The service creator make sure that each request is handled appropriately according to its method. A restful service can serialize data (return data) in various formats like XML or JSON. The data can be handled either by a browser or by a software (service consumer). You can access a restful service directly through a URI (<http://amazon.com/order/123123>) without defining any extra parameters. A restful service provides:

- Uniform interface
- Easy to access
- Interoperability
- Stateless

All methods apart from POST are stateless, they don't change the state of the service. So a not post message doesn't need to know the state of the previous messages, what happened. It will do its thing without the need to know anything from previous messages. This gives among others scalability since you can send the messages through many intermediates, without overloading just one.

- Scalable

There are some frameworks that you can use in order to build APIs that are compatible with the REST architecture, called REST APIs, using django. **Django REST Framework**, **Tastypie** and **django-piston**. So developers will be able to programmatically access your services. With these frameworks you are making your API browsable, available to anyone in the web, through a browser. (Of course you can create software that the user installs and have that software communicate with your API). Creating and exposing APIs allows your web application to interact with other applications through machine-to-machine communication.

The concept is that using such a tool like tastypie, you can easily connect your Django models with the “api” and send the data of the corresponding tables in JSON or XML format.

The first screenshot shows the official YouTube Facebook page at <https://www.facebook.com/youtube>. The second screenshot shows the raw JSON response from the Facebook Graph API at <https://graph.facebook.com/youtube>. The third screenshot shows a filtered JSON response where the 'likes' field has been added to the query string at <https://graph.facebook.com/youtube?fields=id,name,likes>.

```

{
  "id": "7270241753",
  "about": "Discover new channels, watch and share your favorite videos.",
  "can_post": false,
  "category": "product/service",
  "checkins": 29,
  "company_overview": "YouTube provides a forum for people to connect, inform, and in creators and advertisers large and small. .",
  "cover": {
    "cover_id": "10152104891586754",
    "offset_x": 0,
    "offset_y": 0,
    "source": "https://fbcdn-sphotos-f-a.akamaihd.net/hphotos-ak-frc3/v/t1.0-9/s720x0h=0f71b2d5759df58c96719e5c3f7073c&oe=543818528_gda_=1413418562_0a4327b991f2b4472da"
  },
  "founder": "2005",
  "has_added_app": false,
  "is_community_page": false,
  "is_published": true,
  "like_count": 41768558,
  "link": "https://www.facebook.com/youtube",
  "name": "YouTube",
  "page_type": "business",
  "parent_id": "10152104891586754",
  "picture": "https://fbcdn-profile-a.akamaihd.net/hprofile-ak-ppr3/v/t1.0-9/10152104891586754_1413418562_n",
  "talking_about_count": 263847,
  "username": "youtube",
  "were_here_count": 0
}

{
  "id": "7270241753",
  "name": "YouTube",
  "likes": 81793958
}

```

REST APIs

REST APIs work similar with a web site. A client communicates with a server using http. A REST API call is very similar to loading a web page. All the major websites have an api. Google maps API, Twitter API, Instagram API, facebook graph API etc. All these are REST APIs, meaning that they respect some common rules.

Facebook graph api

You can visit the facebook youtube page. If in the url you convert the www to graph then you access the facebook api. You have made an API call. The response in the data of the webpage returned in json format. If you define some parameters in the url you can filter the results of the response.

Googlemaps API

Define the url that you see and you make an api call to the googlemaps api.

You can also make POST requests to APIs, so you can change data in the server using an API, for example making a twit. But to do so you need some authentication. The browser doesn't allow you to put data in the body of the request i.e making a post request, so you have to use a browser extension called POSTMAN REST Client. Many of these APIs are using **oauth** or **oauth2**. Using access tokens (created by the user's id and secret key). The access token passes to twitter and twitter knows that the request comes from you, so it makes the twit from your account.

GoogleMaps

```

{
  "results": [
    {
      "address_components": [
        {
          "long_name": "Chicago",
          "short_name": "Chicago",
          "types": [ "locality", "political" ]
        },
        {
          "long_name": "Cook County",
          "short_name": "Cook County",
          "types": [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name": "Illinois",
          "short_name": "IL",
          "types": [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name": "United States",
          "short_name": "US",
          "types": [ "country", "political" ]
        }
      ],
      "formatted_address": "Chicago, IL, USA",
      "geometry": {
        "bounds": {
          "northeast": {
            "lat": 42.023131,
            "lng": -87.52404399999999
          },
          "southwest": {
            "lat": 41.6443349,
            "lng": -87.9402669
          }
        },
        "location": {
          "lat": 41.8781136,
          "lng": -87.6297982
        }
      }
    }
  ],
  "status": "OK"
}

```

REST is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines. In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture. RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

Restful web services

<http://www.ibm.com/developerworks/webservices/library/ws-restful/>

Summary: Representational State Transfer (REST) has gained widespread acceptance across the Web as a simpler alternative to SOAP- and Web Services Description Language (WSDL)-based Web services. Key evidence of this shift in interface design is the adoption of REST by mainstream Web 2.0 service providers—including Yahoo, Google, and Facebook—who have deprecated or passed on SOAP and WSDL-based interfaces in favor of an easier-to-use, resource-oriented model to expose their services. In this article, Alex Rodriguez introduces you to the basic principles of REST.

In general as I saw the REST is a web architecture that defines the proper communication between the server and the client (via xml or json for example). It defines also some best practices.

A concrete implementation of a REST Web service follows four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.

The article describes these 4 basic principles. Here are some notes regarding the first principle (the explicit HHTP methods):

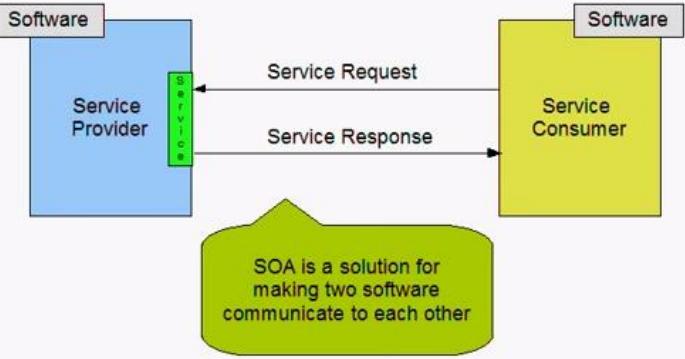
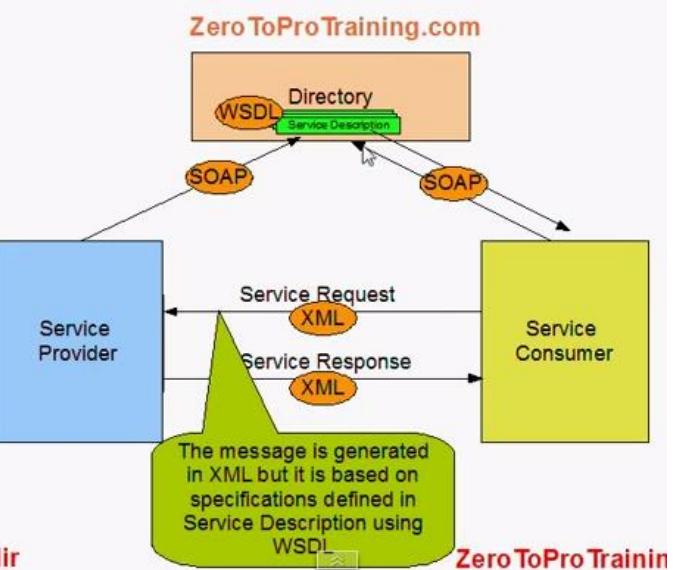
- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

I saw that the request object that is passed to view functions in Django does not differentiates within GET and POST and other one, methods. So probably django does not support the REST web architecture?

➤ Misc

Web service architecture

Web service architecture is used in AJAX calls. It is what is called web 2.0. A standard architecture that describes the communication between software across a network.

 <p>Service oriented architecture SOA</p> <p>It is an architecture that describes how 2 software are communicating with each other through a network. The request is processed by a service in the provider. A service a well defined function that doesn't depend on other services. The sender knows what parameters to send to the service and also what type of respond it wait for.</p> <p>Web services is an implementation of SOA.</p>
 <p>ZeroToProTraining.com</p> <p>Mir ZeroToProTrainin</p> <p>The message is generated in XML but it is based on specifications defined in Service Description using WSDL</p> <p>In web service architecture the service provider (the software that sends the response) publishes in a directory, the description of its service. The consumer can query against this directory to see what services are available and how to communicate with these services.</p> <p>WSDL (web service description language) is an xml based language that is used to create service descriptions (how the service is called, what parameters it expects, what data structure returns etc). The descriptions are saved in a wsdl file. Software can communicate with a WSDL directory using SOAP protocol which is also xml based. The provider to send its description and the consumer to get the description.</p> <p>Based on the description of the service used by the provider the consumer software will create an appropriate XML message to send to this provider.</p> <p>This is the process. First the consumer software communicates using SOAP to take the service description and then formulates its xml message accordingly and sends it to the provider.</p>

SOAP

SOAP is an acronym for Simple Object Access Protocol. It is an XML-based messaging protocol for exchanging information among computers.

- SOAP is a communication protocol designed to communicate via Internet.
- SOAP can extend HTTP for XML messaging.
- SOAP provides data transport for Web services.
- SOAP can exchange complete documents or call a remote procedure.
- SOAP can be used for broadcasting a message.
- SOAP is platform- and language-independent.
- SOAP is the XML way of defining what information is sent and how.
- SOAP enables client applications to easily connect to remote services and invoke remote methods.

Tools per Service

- Load balancer / gateway

[Aws ELB elastic load balancer](#)

- DNS Load balancer

[AWS Route53](#)

- Health check

services like a Heartbeat service (polls other services or/and the services could send messages on their own too)

[Apache zookeeper](#)

- Message queue

[Redis, rabbitMQ, Celery, RQ,](#)

- Rate limiter

- Key value stores

[Redis, etcd, apache zookeeper,](#)

- Lead election

[Etcd, zookeeper](#)

- CDN (for serving static files like img, js etc.)

[AWS CloudFlare, Google CDN](#)

- Authentication / Profiles service

[AWS Cognito](#)

- Session service.

For data that needs to be persistent during a session. In a websockets setup it could store information about which user id corresponds to which connection id. A typical example of a session service or session store, is the basket example. You put items on your basket. You need to have them there even if you reload the page, it is session persisting data

- SQL Databases

[Postgresql, sqlite,](#)

- No sql databases

[MongoDB, AWS DynamoDB, Apache Cassandra](#)

- Distributed File Systems

[Hadoop's HDFS , MapReduce,](#)

- Cache service

[Redis, memcached,](#)

- Analytics service (polls other services for certain metrics and store them)

New relic (performance + errors), sentry (errors), AWS X-tray,

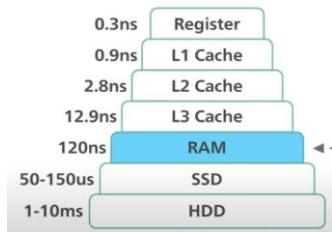
- Group service (in a chat app context for handling chat groups)

Generic services to AWS mapping

Be aware of specific tools for the generic services that you use to design a system. For example the load balancer could be AWS ELB (elastic load balancer), the heartbeat service could be Zookeeper.

Numbers everyone should know

"Numbers Everyone Should Know" from Jeff Dean. Slides #1 , Slide 2			
L1 cache reference	0.5 ns		● Reading speed
Branch mispredict	5 ns		RAM -> 4x faster than -> SSD -> 10x faster than -> 1Gbps connection.
L2 cache reference	7 ns		Reading 1MB from ssd is 4 times slower than reading from memory but 10x faster than reading from 1Gbps network.
Mutex lock/unlock	100 ns		● A relational database could handle a few hundreds of concurrent requests
Main memory reference	100 ns		● A web server of reasonable size, could handle up to 1 million concurrent websocket connections. (a server written in pure Go was able to serve more than a million websockets connections with less than 1GB of ram.)
Compress 1K bytes with Zippy	10,000 ns	0.01 ms	● A good datacenter can have a 5GBps network throughput (5GBps connections between nodes). My 5GHz wifi home network has a throughput of about 100 MBps, so the datacenter has 50x the throughput.
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms	● Maximum on http header values of 8KB on the majority of current servers.
Read 1 MB sequentially from memory	250,000 ns	0.25 ms	
Round trip within same datacenter	500,000 ns	0.5 ms	
Disk seek	10,000,000 ns	10 ms	
Read 1 MB sequentially from network	10,000,000 ns	10 ms	
Read 1 MB sequentially from disk	30,000,000 ns	30 ms	
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms	
Where			
• 1 ns = 10^{-9} seconds			
• 1 ms = 10^{-3} seconds			



a Linux Thread needs 8MB of memory

each allocated thread reserves 8 MB of memory for its stack by default. You can set the default stack size to 1MB. (for 10k concurrent users you need 10GB of memory if you have one thread per connection)

1 server per 1000 users

With 500,000 servers, Twitter has roughly 1 server per 1,000 users.

The C10k challenge

At the start of the 21st century, engineers ran into a scalability bottleneck: web servers were not able to handle more than 10,000 concurrent connections.

When he asked this in 1999, for many server admins and engineers, serving 10,000 concurrent visitors was something that would be solved with hardware. The notion that a single server on common hardware could handle this type of CPU and network bandwidth without falling over seemed foreign to most.

You can overcome this problem without really solving it

- with one thread per connection services you must use load balancing to balance the load among servers (this doesn't solve the underlying problem of handling more than 10k concurrent users per server, but overcomes it)
- with single thread services (event loop and coroutines)

Concurrency does not solve the C10k problem in itself, but it absolutely provides a methodology to facilitate it. The most notable and visible example of an approach to the C10K problem today is Nginx, which was developed using concurrency patterns, widely available in C by 2002 to address—and ultimately solve—the C10k problem.

Using aws lambdas

Mock interviews

Amazon, Design Parking Garage

1. First he clarifies the product requirements
2. Then defines the API Endpoints
3. Then Data schema
4. Finally the Architecture

In endpoints definition you also define the arguments and the return

In database schema you also define the data type for each field.

Exponent Interview

Product Requirements

- Need to be able to reserve a parking spot and receive some kind of ticket or receipt
- Need to be able to pay for a parking spot
- System needs to have a high consistency (no two people should be able to reserve the same spot at the same time)
- 3 types of vehicles (compact, regular, and large)
- flat rate based on time, but different rates depending on the type of parking

Public Endpoints

/reserve

Params: garage_id, start_time, end_time

Returns: (spot_id, reservation_id)

/payment

Params: reservation_id

Note: Likely using an existing API to handle (Stripe, Square, etc...)

/cancel

Params: reservation_id

Internal Endpoints

/calculate_payment

Params: reservation_id

/freespots

Params: garage_id, vehicle_type, time

/allocate_spot

Params: garage_id, vehicle_type, time

/create_account

Params: email, password, (optional) first_name, (optional) last_name

Note: smaller vehicles can fit into larger spots if necessary and therefore should be included in the overall number of spots

/login

Params: email, password

Exponent Interview

Data Schema

Reservations

Garage

id	primary key, serial
garage_id	foreign key, int
spot_id	foreign_key, int
start	timestamp
end	timestamp
paid	boolean

id	primary key, serial
zipcode	varchar
rate_compact	decimal
rate_reg	decimal
rate_large	decimal

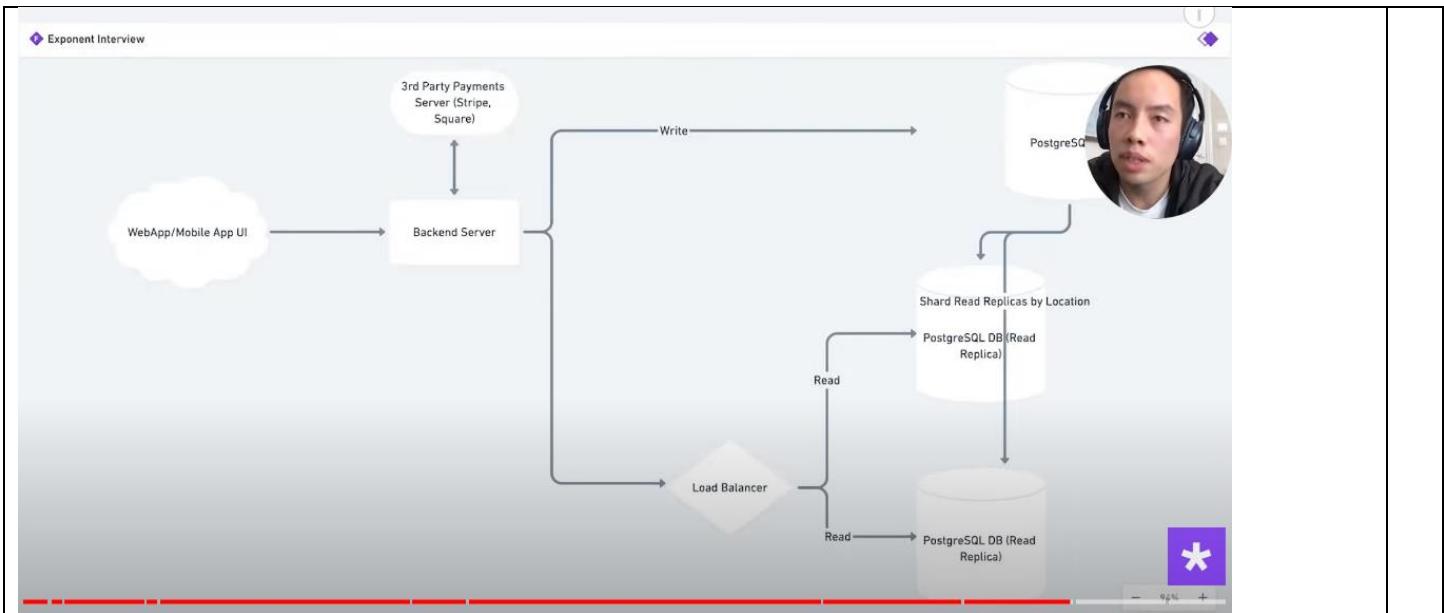
Spots

Users

id	primary key, serial
garage_id	foreign_key, int
vehicle_type	enum
status	enum

id	primary key, serial
email	varchar
password	varchar (note that this is probably SHA-256 hash)
first_name	varchar
last_name	varchar

Vehicles



Design Netflix

Have in mind the user metadata and video metadata data. Videos are stored in blob store like S3 while video metadata on RDBMS (or noSQL) and you do an additional choice for user metadata too (SQL or noSQL).

User metadata

Watched videos, last watched timestamp, likes.

They are 100 kb per user times 200m users = 20 TB

Video metadata

Name of video, description, category,

Tips

Architecture tips

- What is more important, consistency or availability in this app?

In this case consistency. You don't want two users reserving the same spot. So you choose strong consistency.

- Is it a read or write heavy application?

It is a read heavy application. So we can have one write db and a few read replicas with strong consistency instead of eventual consistency. Notice that we can shard the read replicas by location. Each location (Europe) has one read replica. To ensure consistency you can use as many people do, read locks for the read dbs. When there is a write, the read replicas are read locked. When the write is done and propagated, the read lock is released. If you have sharding by location on the read replicas where each location is served by a local read replica which contains only the data from its region, then you can readlock only the local replica for which there is a write.

Field tips

- Location

When you want to store the location of an entity you can use the zipcode of each location. Its quite generic and convenient. The extended zip code contains 4 additional digits in the form of 55236-1234. It has a hyphen so the type is varchar not int. In this case a garage has a zip code field. Then you can do sharding based on location if you want

- Enums

A variable with multiple choices can be stored as enum in the db instead of as a simple charvar type. The enum is stored as integer (1, 2, 3 for three choices). This is more efficient but in postgres you can't remove a choice later (you can add though). Just a trade off to have in mind.

- Payment amounts

Store things in cents (smallest currency unit you're dealing with) and save yourself some hassle. If cents are not granular enough use a big multiplier to create larger integers, I recommend something like micro-dollars where dollars are divided by 1 million.

<https://rietta.com/blog/postgresql-currency-types/>

Your choices are:

- bigint: store the amount in cents. This is what EFTPOS transactions use.
- NO decimal(15,4) : store the amount with exactly four decimal places. This what most general ledger software uses.
- NO float : terrible idea - inadequate accuracy. This is what naive developers use.

Take this as an example: 1 Iranian Rial equals 0.000030 United States Dollars. If you use fewer than 5 fractional digits then 1 IRR will be rounded to 0 USD after conversion. I know we're splitting rials here, but I think that when dealing with money you can never be too safe.

Postgresql has a money type field. Isn't the most convenient option.

Primary key

Specify it's "subtype". For example serial pk, which is automatically increased by one at each insertion.

Tips

RunwayML

A peek at our technical stack

The rich UI of our video editing and collaboration tools is powered by Typescript and React/Redux, while the real time compositing and graphics engine behind our interactive preview runs on WebGL2 and WebAssembly. Our video streaming backend components are written in Python, use a lot of FFmpeg/libav and HLS for on-the-fly transcoding, PyTorch and TorchScript for ML inference, and are deployed as containerized services on Kubernetes. Our API endpoints for real-time collaboration and media asset management are written in Typescript and node.js and are deployed as serverless functions on AWS Lambda.

Code review

Easy to read and comments, scalable, maintainable, modular, non-redundant

As a code submitter

- Reduce the number of lines you send for code review (10-100 lines). Don't send 1000 lines of code in one pull request
- Leave good context for your code with comments
- If there is a change in another part of the codebase related to your main changes, describe why you changed this to your code reviewer

As a code reviewer

- Make clear when something is just a suggestion that you don't feel strongly about and something that you do. You can add a "use your judgement" comment for example.

- As a team make clear rules on when you are rejecting a pull request to avoid bad feelings
- Discuss minor matters in person and not in the pull request process to save time.

Questions to ask on a design X task

1. Do we have to design the whole system or a specific part of it or Create a list the basic features I'm thinking and ask for feedback
2. How many 9s of availability do we need (highly available or 2-3 nines are ok)
3. How many users do we have to support?
4. Are users spread all over the world or they are in a specific region?
5. How often we make changes to the content of the offered service?

Architecting a Cloud Native API Solution

<https://www.youtube.com/watch?v=sKfep-UmZeM> nice 10mins video

infrastructure as code, pipelines, dev/test/stage/prod environments, Kubernetes cluster, logging, monitoring, openshift.

The System Design Primer

There is a github repository “The System Design Primer” which has a lot of useful content.

A reddit post (api gateway + lambdas vs drf)

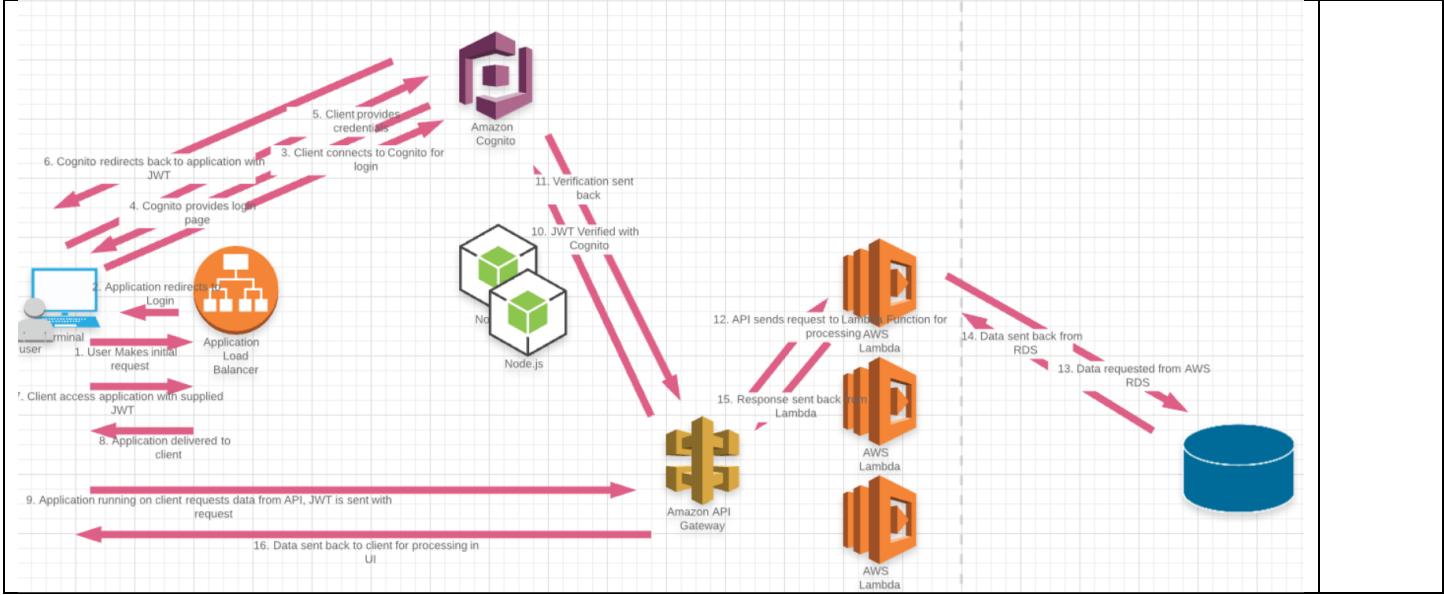
I am looking to build out the backend for a new application that will serve 10s/100s of customers with 1-20 employees each, so not a huge user count. The application is very much data driven so quick access to the data is important, but fast development is equally important.

Frontend will most-likely by ViewJS connecting to the APIs. I prototyped the APIs in DRF and we'll be hosting in AWS so I started researching how to architect this and I came up with a lot of possibilities, the problem with having many options.

We originally were thinking DRF with ELB in front and RDS in the back. I thought about trying to simplify, and provide modular updates, by using API Gateway with Lambda. I am aware of the latency this causes, although I haven't been able to pin down consistent numbers which is also a problem with Lambda. Instead of needing DRF I could use basic python functions to respond to the API requests and have the Gateway perform everything else along with Cognito providing the authentication. Cognito would allow us to easily integrate with OAUTH and SAML. **No better not do it like this (one lambda per endpoint). Instead have one lambda per Django rest framework service. That simplifies deployment as there's only one lambda to worry about.**

I keep going back and forth on which would provide the best overall "value" when taking speed of deployment, scalability (500ish daily users) and resiliency into account.

Here is an initial plan (the initial steps should go through to the Node.js



Twitter

https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale

With 500,000 servers, Twitter has roughly 1 server per 1,000 users. Only about 20% is the key-value store that actually stores the tweets. A huge chunk is cache servers, and another huge chunk is Hadoop for analytics. talked to a twitter engineer about it once. apparently a lot of the compute goes to a tiny minority of "whales", compulsive twitter users that check the site 5 or even 10 times PER DAY.

Using a relational data store does make it easier to design features like allowing users to edit posts (something Twitter would find it hard to support because of the sharding in its distributed data store) and running relational databases is a well-known task

Stackoverflow

<http://highscalability.com/stack-overflow-architecture> 2009

StackOverflow famously runs on three servers total (2 web, one database)

A site like Stack Overflow with over 100 million monthly visitors can run on a relatively small number of servers (somewhere between 20 and 50 servers for serving the site itself, although with more hardware for networking, logging, monitoring, backup and other key infrastructure tasks). (Quinn notes that Stack Overflow also has built out a CDN, so it's not accurate to think of it being served from a single location on a handful of servers.)

Wikipedia is just a couple hundred servers, but of course they are almost write-only which makes it easier

Behavioral interview

Humility, respect, team player, being able to work under pressure, to handle conflicts, to learn from mistakes, to show initiative, to show leadership

CDN for static files (and content stored as files)

<https://www.youtube.com/watch?v=QEGo6ZoN-ao> nice 5 mins

You can set up your static files which are stored in a blob store like S3, to be served by a CDN. So your react app js bundle and index.html could be served by the CDN and then when the user interacts with the page it touches your API. Have also in mind that any static content in your website (like the algoexpert questions with the default code answers) could be stored as text files in a blob store.

Start thinking in microservices terms

I sense that a good way to start thinking when designing a system, is to think of the main features and then think of them initially as distinct microservices. As you continue working on them, you might decide to combine some of them in one service or even combine all of them in one monolith. The point is that it is easier to start thinking about the system in microservices terms.

Think about resilience and scalability

No single points of failure. Know how you can make your distinct services resilient.

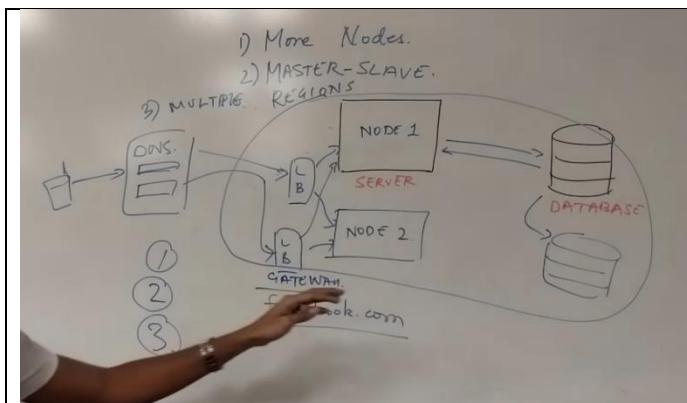
- Databases

Master/slave architecture (preferred), sharding (complicated), no sql databases

- Random services

You will require load balancer to distribute the load between them.

System resiliency



Avoid single points of failure.

For databases this might require a master slave architecture. For services this might require to have more than one nodes that run the same service. This means you need a load balancer (a gateway). Then you need to avoid gateway risk as a single point of failure so you want multiple gateways. Each gateway has its own IP so this means that you need to define more than one ips for your domain name in the DNS servers so that they route requests to all your gateways.

And then you might need to distribute your system across different regions to secure it against major region wide failures.

Overloading servers solutions

Problems:	Solutions:
1. Cascading Failure	1. Rate Limiting
2. Going viral	2. Pre Scale
3. Predictable load increase	3. Auto Scale
4. Bulk Job Scheduling	4. Batch Processing
5. Popular Posts	5. Approximate Statistics

Consistency versus availability

There is a trade off when you choose. More of the first, less of the other. For example if you have to write to three databases, what happens when one database fails? You have to make a choice. You either return an error because the data was not written in all three (this means reduced availability but strong consistency), or you return successfully and you deal with the issue later (eventual consistency).

Health check

Some metrics a service could publish for health check reasons: number of processes, request queue size, number of requests, error rate. These metrics could be read by an anomaly detection service that could notify engineers in case of anomalies.

Microservices

Intro

Some Key things

- Inter Services communication to enforce referential integrity for example FK relationships (because now you have many dbs not just one that could enforce referential integrity on its data). It is better done with events instead of api communication. You need a publish subscribe system. Services publish events to channels where other services are subscribed to. For example user-delete event. The other services take that event and act accordingly, for example delete user related data. The event can be json data or anything else. The services parse it to native datatypes and use it. You implement such a mechanism with an event streaming platform (or publish-subscribe platform) like Kafka or google cloud Pub/Sub.
- Authentication. You have an auth service, yours or 3rd party. You get a jwt from that service with the permissions for all services. If not for all services then you do token exchange and you get an additional jwt with the specific service permissions.
- Client libraries. As I saw in most large architectures with many services, they create client libraries (api wrappers) so that other services communicate with a service using its client library that they have installed, instead of with raw rest calls. There are pros and cons.
- API gateway.
- Identity provider service
- Service registry and discovery service

Example of event driven architecture

Consider two services: Notification and User. Assume that the Notification Service has ownership of the Notification table, and the User Service has ownership of the User table. Suppose the Notification Service has generated a notification with the Notification Status "New" and published a "Notification Created" event. This event will be consumed by Email Service, the notification status will be changed to "Processing" and a "Send Notification" event will be published. The Notification Service then consumes the "Send Notification" event and changes the notification status to "Processed." This is a simple example of how event-driven services work asynchronously.

A typical difficulty

One such problem is figuring out how to perform business transactions that span several systems while maintaining data integrity. To ensure consistency in a typical monolithic application, you might want to use ACID transactions. However, putting this into practice in a microservices application is not an easy task.

Event driven patterns

The following patterns are utilized in the event-driven manner of developing microservices: Event Stream, Event Sourcing, Polyglot Persistence, and Command Query Responsibility Separation (CQRS).

They are designed to do one thing very well. Each microservice has exactly one well-known entry point. While this may sound like an attribute of a component, the difference is in the way they are packaged.

Microservices are not just code modules or libraries – they contain everything from the operating system, platform, framework, runtime and dependencies, packaged as one unit of execution.

Each microservice is an independent, autonomous process with no dependency on other microservices. It doesn't even know or acknowledge the existence of other microservices.

Microservices communicate with each other through language and platform-agnostic application programming interfaces (APIs). These APIs are typically exposed as Rest endpoints or can be invoked via lightweight messaging protocols such as RabbitMQ. They are loosely coupled with each other avoiding synchronous and blocking-calls whenever possible.

As Janakiram points out, such communication should be lightweight and platform-agnostic, what Lewis and Fowler refer to as dumb pipes.

Essentially, microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

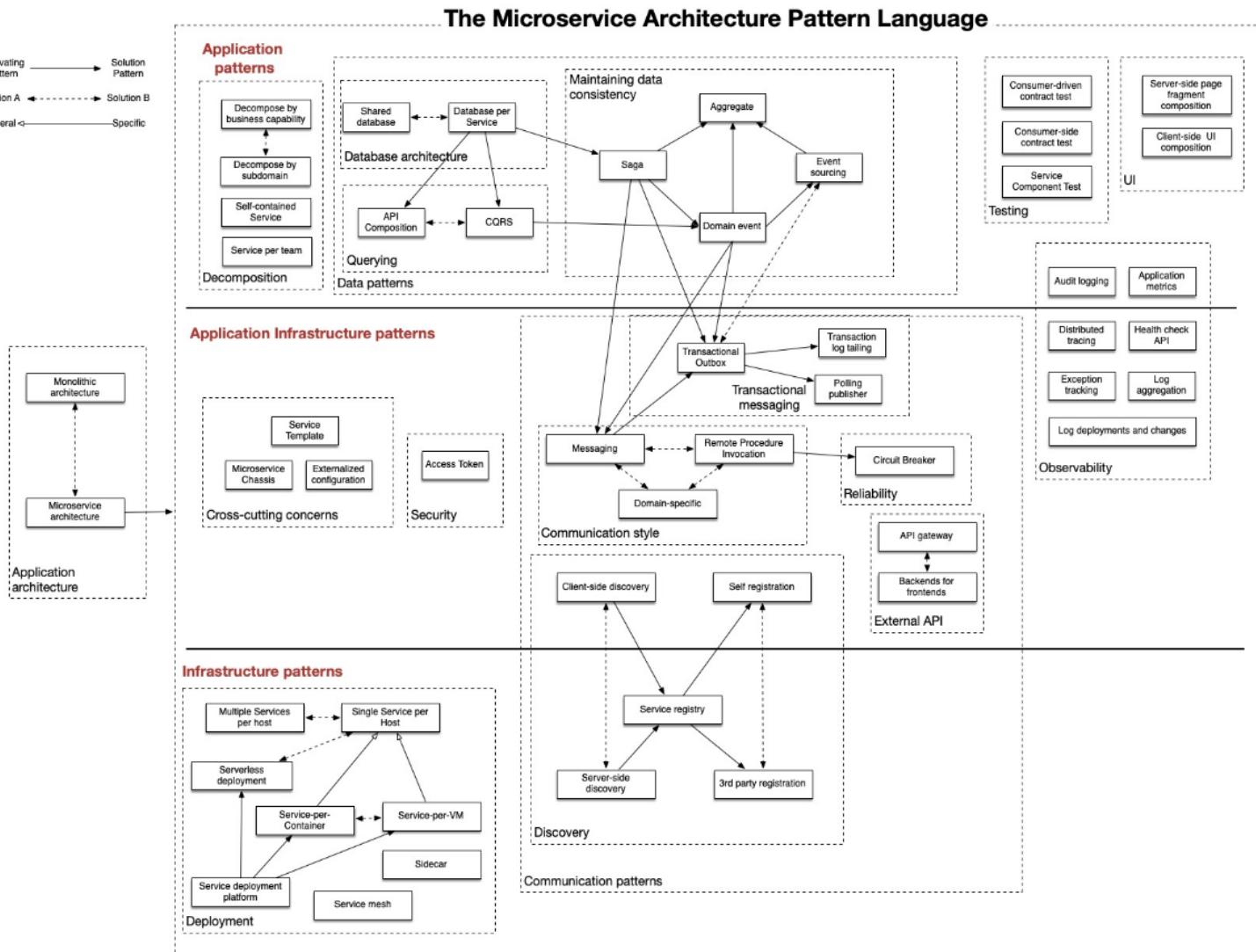
How the services communicate with each other depends on your application's requirements, but many developers use HTTP/REST with JSON or Protobuf. DevOps professionals are, of course, free to choose any communication protocol they deem suitable, but in most situations, REST (Representational State Transfer) is a useful integration method because of its comparatively lower complexity over other protocols.

Microservices frequently use NoSQL or micro-SQL databases (which can be connected to conventional databases).

Microservices vs the more traditional Monolithic Architecture pattern.

Design Patterns of Microservices

<https://microservices.io/patterns/index.html> excellent descriptions



Data management

How to maintain data consistency and implement queries?

- Database per Service - each service has its own private database
- Shared database - services share a database
- Saga - use sagas, which are sequences of local transactions, to maintain data consistency across services
- API Composition - implement queries by invoking the services that own the data and performing an in-memory join
- CQRS - implement queries by maintaining one or more materialized views that can be efficiently queried
- Domain event - publish an event whenever data changes
- Event sourcing - persist aggregates as a sequence of events

Transactional messaging

How to publish messages as part of a database transaction?

- Transactional outbox
- Transaction log tailing
- Polling publisher

Communication patterns

Style

Which communication mechanisms do services use to communicate with each other and their external clients?

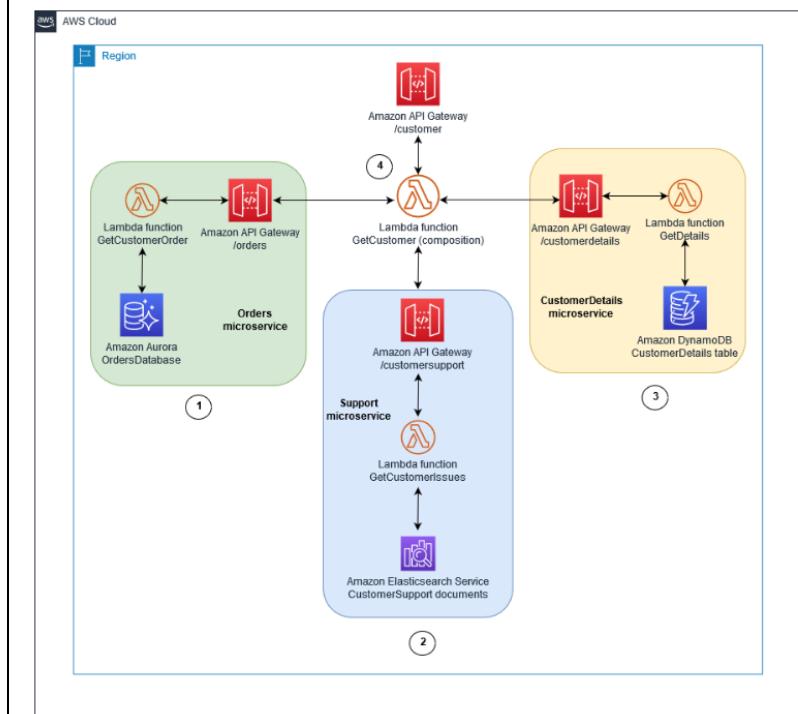
- Remote Procedure Invocation - use an RPI-based protocol for inter-service communication
- Messaging - use asynchronous messaging for inter-service communication
- Domain-specific protocol - use a domain-specific protocol
- Idempotent Consumer - ensure that message consumers can cope with being invoked multiple times with the same message

Querying data on multiple microservices

Api composition

The API composer, or aggregator, is a separate service, that implements a query by invoking individual microservices that own the data. It then combines the results by performing an in-memory join. API composition functionality could be offered by API gateways (not by aws api gateway as of 2020). It could also be implemented by a lambda function.

An aws implementation of the API composition pattern



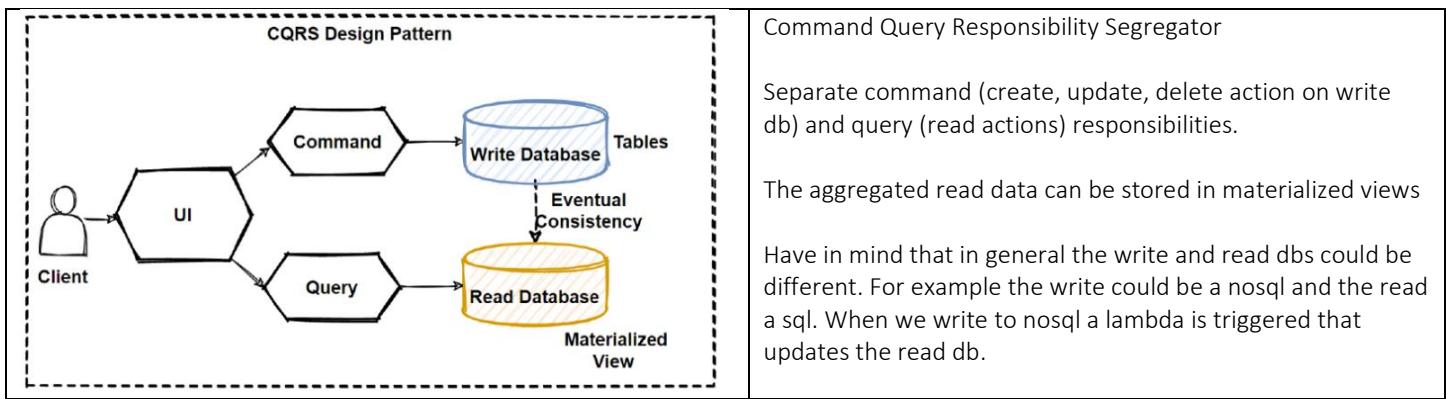
Pros

We can abstract away the internal complexity and expose a unified api.

Cons

- It might not be suitable for complex queries and large datasets that require in-memory joins.
- Your overall system becomes less available if you increase the number of microservices connected to the API composer.
- Increased database requests create more network traffic, which increases your operational costs.

CQRS



Distributed transactions (sagas recommended)

https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture#what_is_a_distributed_transaction excellent overview

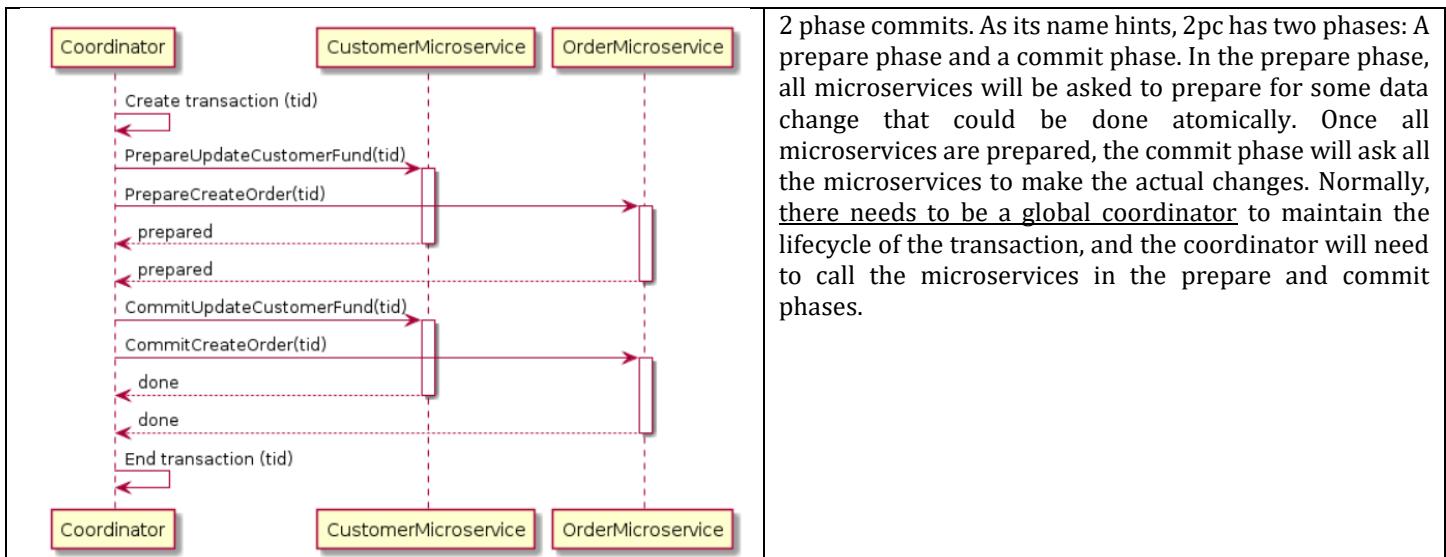
An operation consisted of many local transactions on various microservices. It is not a real transaction, in the sense that it is not ACID. It can't be rolled back automatically on error. You have to implement this logic yourself. Two problems we need to address with distributed transactions:

1. all steps complete or no steps complete.
2. If an object is written by a transaction and at the same time (before the transaction ends), it is read by another request, should the object return old data or updated data?

There are two main patterns for distributed transactions. 2 phase commits and Sagas. Saga is the recommended approach

2pc

- Synchronous updates (strong consistency)
- But not recommended because it can lock data for a long time (if a transaction is slow, for example payments).

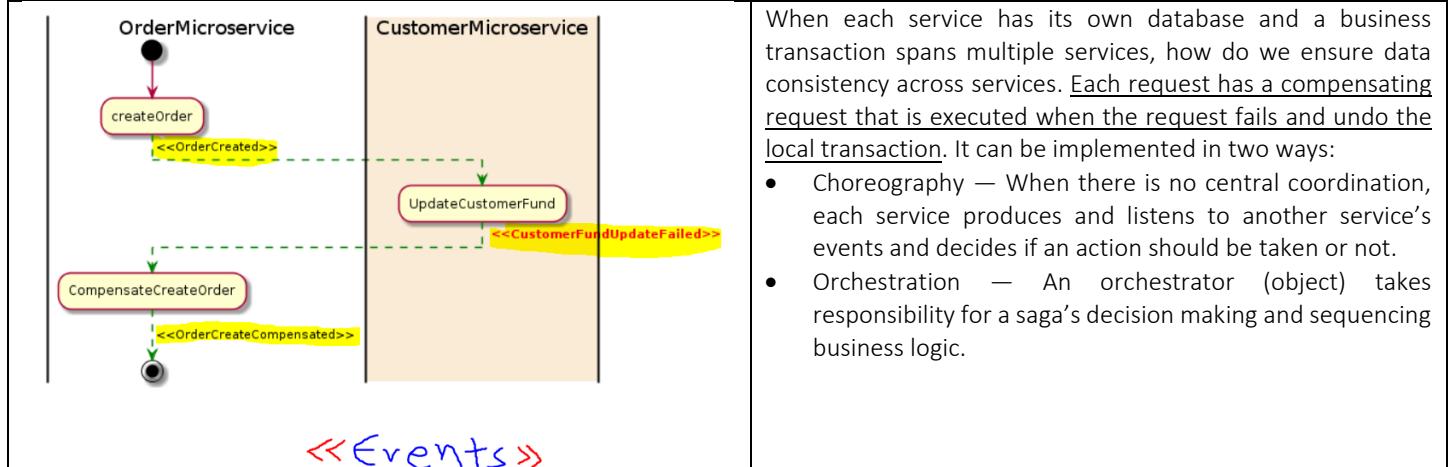


Saga

Asynchronous, recommended (it uses events and compensated actions).

Cons

- The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. To address the complexity issue of the Saga pattern, it is quite normal to add a process manager as an orchestrator. The process manager is responsible for listening to events and triggering endpoints
- Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.



A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

There are also the following issues to address:

- In order to be reliable, a service must atomically update its database and publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below.
- A client that initiates the saga, which an asynchronous flow, using a synchronous request (e.g. HTTP POST /orders) needs to be able to determine its outcome. There are several options, each with different trade-offs:
 - The service sends back a response once the saga completes, e.g. once it receives an OrderApproved or OrderRejected event.
 - The service sends back a response (e.g. containing the orderID) after initiating the saga and the client periodically polls (e.g. GET /orders/{orderID}) to determine the outcome
 - The service sends back a response (e.g. containing the orderID) after initiating the saga, and then sends an event (e.g. websocket, web hook, etc) to the client once the saga completes.

Flow's microservice architecture

Design microservice architectures the right way, (Flow company)

<https://www.youtube.com/watch?v=j6ow-UemzBc&list=WL&index=23&t=2s>

API definitions

Notice: they are in their own repo. They don't live in the service's repo. If you want to change an api definition you have to modify the json object that describes it. and then commit your change. They have CI/CD on that repo too. they run tests on the json file with linter, checking if the syntax is within their standards. Then you can implement the change in the service's repo.

API Definition

Done correctly, even things like GDPR compliance can be modeled

```
"user": {  
  "description": "Represents a single user in the system",  
  "fields": [  
    { "name": "id", "type": "string" },  
    { "name": "email", "type": "string", "required": false, "annotations": ["personal_data"] },  
    { "name": "name", "type": "name", "annotations": ["personal_data"] },  
    { "name": "status", "type": "user_status", "default": "active" }  
  ]  
},  
  
"user_form": {  
  "fields": [  
    { "name": "email", "type": "string", "required": false, "annotations": ["personal_data"] },  
    { "name": "password", "type": "string", "required": false, "annotations": ["personal_data"] },  
    { "name": "name", "type": "name_form", "required": false, "annotations": ["personal_data"] }  
  ]  
},
```

They start with API definitions. This is the first job. They define a resource, here the user resource and completely describe it (but where is the password?)

An instance of that resource can be generated by an object named by convention "resource-name_form", so here user_form.

Resource Oriented

```
"resources": {  
  "io.flow.common.v0.models.user": {  
    "operations": [  
      {  
        "method": "GET",  
        "description": "Returns information about a specific user.",  
        "path": "/:id",  
        "responses": {  
          "200": { "type": "io.flow.common.v0.models.user" },  
          "401": { "type": "unit" },  
          "404": { "type": "unit" }  
        }  
      },  
  
      {  
        "method": "POST",  
        "description": "Create a new user. Note that new users will be created with a status of pending and will not be able  
Flow team.",  
        "body": { "type": "user_form" },  
        "responses": {  
          "201": { "type": "io.Flow.common.v0.models.user" },  
          "401": { "type": "unit" },  
          "422": { "type": "io.flow.error.v0.models.generic_error" }  
        }  
      }  
    ]  
  }  
}
```

This is how they expose a resource (defining how it can be interacted with, through an API).

We have a user model. how we interact with it? by defining the api operations on it.

We expose the user model as a resource, making it available through an API.

During the API design phase, (no code written yet) they check if a change to the api is breaking or not. If it is they have to decide in the early phase if they will implement it or if they will use a workaround. They make this test with a tool called API builder. An open source Tool created by them, for end to end API design

API implementation

The API builder, contains code generation functions they have created. They create code from the API definition files. Not the entire implementation but some fundamental stuff arising from the api definition. They generate 3 things: a Routes file, a Client library, a mock client.

Code Generation: Routes

```
GET   /users/:id           controllers.Users.getById(id: String)  
POST  /users               controllers.Users.post()  
  
      description : Represents a single user in the system  
      fields :  
  
Guarantee that API operations are actually defined  
User friendly paths  
Consistent naming for methods
```

A routes example in their web framework (they use scala).

Automated generation of a client library is also quite cool.

The mock client is a library that returns the objects the api specification defines, without actually calling any service. They can be used for testing. They write unit and integration tests against the mock clients.

Code Generation: Client	Code Generation: Mock Client
<pre> 867 override def post(868 userForm: io.flow.user.v0.models.UserForm, 869 requestHeaders: Seq[(String, String)] = Nil 870)(implicit ec: scala.concurrent.ExecutionContext): scala.concurrent.Future[io.flow.common.v0.mod... 871 val payload = play.api.libs.json.Json.toJson(userForm) 872 873 _executeRequest("POST", s"/users", body = Some(payload), requestHeaders = requestHeaders).map { 874 case r if r.status == 201 => { 875 _root_.io.flow.user.v0.Client.parseJson(876 "io.flow.common.v0.models.User", r, _.validate[io.flow.common.v0.models.User] 877) 878 } 879 case r if r.status == 401 => throw io.flow.user.v0.errors.UnitResponse(r.status) 880 case r if r.status == 422 => throw io.flow.user.v0.errors.GenericErrorResponse(r) 881 case r => throw io.flow.user.v0.errors.FailedRequest(882 r.status, s"Unsupported response code \${r.status}. Expected: 201, 401, 422" 883) 884 } 885 } </pre>	<pre> 101 def post(102 userForm: io.flow.user.v0.models.UserForm, 103 requestHeaders: Seq[(String, String)] = Nil 104)(implicit ec: scala.concurrent.ExecutionContext): scala.concurrent.Future[io.flow.common.v0.mod... 105 scala.concurrent.Future.successful(106 io.flow.common.v0.mock.Factories.makeUser() 107) 108 } </pre>

Mock and Client from Same Source

Enables High Fidelity, Fast Testing

then they write code for api

Now Let's Implement	fundamental stuff. On post requests, validate and create.
<pre> def post(): Action[JsValue] = Anonymous.async(parse.json) { request => Future { usersDao.create(request.user, request.body.as[UserForm]) match { case Left(errors) => { UnprocessableEntity(Json.toJson(Validation.errors(errors))) } case Right(newUser) => { Created(Json.toJson(newUser)) } } } } </pre>	

Goal: Code we actually write is simple, consistent

Service's storage requirements

<p>Database Architecture</p> <p>Each micro service application owns its database</p> <p>No other service is allowed to connect to the database</p> <p>Other services use only the service interface (API + Events)</p>	<p>Other services communicate with the service's database either through the service's interface (service's API or through events).</p>
<p>Create a Database</p>  <pre>\$ dev rds --app test Confirm settings: - db_name: testdb - storage: 100 - db_instance_class: db.t2.medium - db_instance_id: testdb20180623 Proceed? (y/n):</pre>	<p>They have created a CLI called dev that all devs use for common tasks. Here for creating the db for your service. It uses the company's conventions. The db experts wrote the create db command for the cli and all other devs just use that. They don't login into amazon to create dbs etc. Everything is automated.</p> <p>They also generate the db tables with auto code generation from metadata that describe the table.</p>

Testing

Unit Tests for mock clients are automatically generated too. for example. testing the GET method on /users/:id it calls the createuser function. It expects a user object back. This user object is a mock user object created automatically. If It passes an invalid id it expects a 404.

The mock user object

```
def identifiedClient(
  user: UserReference = testUser
): Client = {
  new Client(
    wsClient,
    s"http://localhost:$port",
    defaultHeaders = authHeaders.headers(AuthHeaders.user(user))
  )
}
```

Test Resource Operations

```
"GET /users/:id" in {
  val user = createUser()
  await(
    identifiedClient().users.getById(user.id)
  ) must equal(user)
}

"GET /users/:id w/ invalid id returns 404" in {
  expectNotFound(
    identifiedClient().users.getById(UUID.randomUUID.toString)
  )
}
```

Use the generated mock clients to write simple tests

Continuous deployment

They wrote a tool for handling CD called “delta”. A deployment is initiated when a new tag is applied to the master branch in github. So, a new commit to master. It has a tag like 0.3.12. github sends a webhook to delta. Delta checks the current deployed tag (the current head of master). It is different. It sets the deployment’s desired state to the new tag. This triggers jobs. Delta polls. Is the docker image ready? If yes, deploy it in ECS. It has a dashboard with all microservices.

Continuous Delivery

Deploy triggered by a git tag
Git tags created automatically by a change on master (e.g. merge PR)
100% automated, 100% reliable

Flow/user is a microservice.

Auto Deploy on New Commit on Master

Build	Desired state last set	State
flow/user	seconds ago	Transitioning from 0.4.54 (2) to 0.4.55 (2)

Desired State

0.4.55

2 instances
Updated seconds ago

Last State

0.4.54

2 instances
Updated seconds ago

Microservice infrastructure

Microservice Infrastructure – keep it simple

```
1 builds:
2   - root:
3     instance.type: t2.small
4     port.container: 9000
5     port.host: 6021
6     version: 1.3
```

Infra specialists know the details. Devs just define a simple yaml file with the most basic infra requirements for their service.

Health check

```
$ curl --silent https://user.api.flow.io/_internal_/healthcheck | jq .
{
  "status": "healthy"
}

{
  "models": {
    "healthcheck": {
      "fields": [
        { "name": "status", "type": "string", "example": "healthy" }
      ]
    }
  },
  "resources": {
    "healthcheck": {
      "path": "/_internal_",
      "operations": [
        {
          "method": "GET",
          "path": "/healthcheck",
          "responses": {
            "200": { "type": "healthcheck" },
            "422": { "type": "io.flow.error.v0.models.generic_error" }
          }
        }
      ]
    }
  }
}
```

It is an endpoint that all services have. Another service calls that.

Here there is just a check that checks if the service is available. You can implement more checks:

- Has the service Access to the db
- Env vars are available

If not healthy, it is never put into traffic

Events

Consume events by default (not API)

<p>"We have an amazing API, but please subscribe to our event streams instead."</p>	<p>For their internal applications they use almost completely event streams instead of API calls. They use API calls only for things that need to happen synchronously. Otherwise they consume events and process things asynchronously.</p>
--	--

<p>Principles of an Event Interface</p> <p>First class schema for all events</p> <p>Producers guarantee at least once delivery</p> <p>Consumers implement idempotency</p> <p>Flow:</p> <ul style="list-style-type: none">- End to end single event latency ~ 500 ms- Based on postgresql – scaled to ~1B events / day / service	<p>You have to have Events' schema</p> <p>They use auto code generation for the code that produces and consumes events. Code is created by the events schemas. Ideally the event schema is in binary format (as I understood because you can't read it, it forces you, the dev, to use the code generation to produce and consume an event)</p> <p>They chose this standard</p> <ul style="list-style-type: none">• At least once delivery. This means the following:• Consumers must implement idempotent event consuming operations because the event could be delivered more than once. <p>They build their system on top of aws kinesis (Amazon Kinesis makes it easy to collect, process, and analyze real-time, streaming data)</p>
<p>Events: Approach</p> <p>Producers:</p> <ul style="list-style-type: none">• Create a journal of ALL operations on table• Record operation (insert, update, delete)• On creation, queue the journal record to be published• Real time, async, we publish 1 event per journal record• Enable replay by simply requeueing journal record	<p>Producer is a service that creates/produces an event</p> <p>For every table, there is a <u>journal-table</u> that corresponds to it, that contains all the operations performed on that table. So there is the user table and the user_journal table. The journal keeps track of all operations applied to the table.</p> <ol style="list-style-type: none">1. You insert a user in the user table2. You add an insert operation to the journal_table.3. Add the journal event to the microservice's "local" queue to be published4. Notify an actor that something has changed (I understand actors as workers that listen to that queue)5. Actor comes in, does some work and publishes the event to kinesis. <p>Consumer.</p> <p>Stores the event in a local db for a certain retention period.</p> <p>Add it to its local queue to be consumed</p> <p>A worker process it the events in microbatches.</p>
<p>Events: Approach</p> <p>Consumers:</p> <ul style="list-style-type: none">• Store new events in local database, partitioned for fast removal• On event arrival, queue record to be consumed• Process incoming events in micro batches (by default every 250ms)• Record failures locally	<p>If there is a failure in the consumer, it fixes the bug, and because it has the event in its local storage, it doesn't have to go back to kinesis to get the event again. It gets it from its db and requeues it. this process is entirely local.</p>

<p>Events: Schema First</p> <p>1 model / event</p> <p>N events in one union type</p> <p>1 union type / stream</p> <p>Stream owned by 1 service</p> <p>Most services define exactly 1 stream</p>	<pre>"unions": { "user_event": { "discriminator": "discriminator", "types": [{ "type": "user_upserted" }, { "type": "user_deleted" }] } }, "models": { "user_upserted": { "fields": [{ "name": "event_id", "type": "string" }, { "name": "timestamp", "type": "date-time-iso8601" }, { "name": "user", "type": "io.flow.common.v0.models.user" }] }, "user_deleted": { "fields": [{ "name": "event_id", "type": "string" }, { "name": "timestamp", "type": "date-time-iso8601" }, { "name": "user", "type": "io.flow.common.v0.models.user" }] } }</pre>	<p>The union “user_event” here has two events. Every union type maps to a single stream in kinesis. Also, each union_type corresponds to one microservice.</p> <p>The user_deleted event is an object with three fields.</p>
--	--	--

Partitioning is the database process where very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables, queries that access only a fraction of the data can run faster because there is less data to scan. You can partition a table by date (range partitioning in postgresql).

<p>Producers: Database Journal</p> <p>Document retention period</p> <p>Code generate journal</p> <p>Use partitions to manage storage</p>	<pre>"journal": { "interval": "daily", "retention": 3 }</pre> <p>https://github.com/gilt/db-journaling</p>	<p>They describe the storage requirements with metadata as he already told. Journals is a new storage requirement. They describe it too. they define the interval period and the retention. This definition will create daily partitions in the db and retain the last 3 days.</p> <p>The dev creates the definition and the journal tables are created automatically.</p>
---	--	--

They generate automatically tests for event publishing and consuming. You publish an event. You expect it to be written in your db. Or, we create a user. Eventually the user event should exist in the queue.

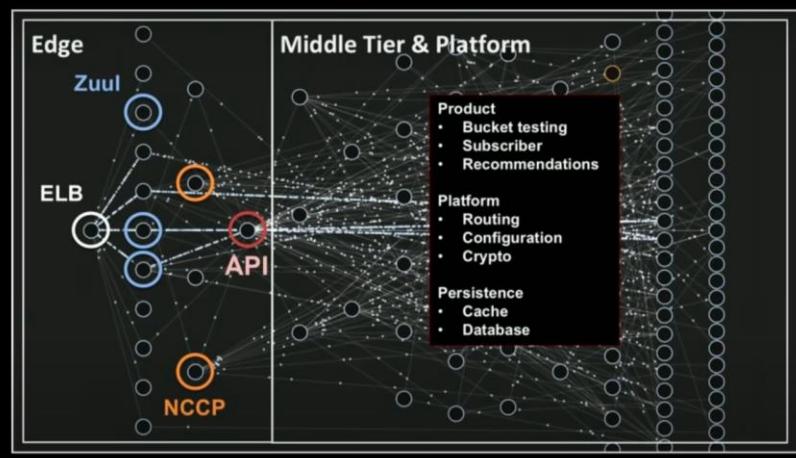
Netflix mastering chaos

Mastering Chaos - A Netflix Guide to Microservices

<https://www.youtube.com/watch?v=CZ3wluyvHeM> watch this again. Very useful. Important notes to be extracted.

Netflix has an open source software center. It has open sourced many tools like Eureka and Zuul.

Mastering Chaos - A Netflix Guide to Microservices

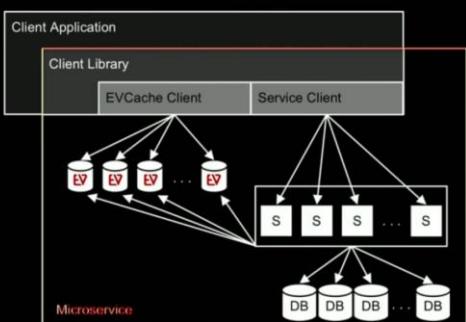


Eureka and Zuul work in tandem to provide service discovery and load-balanced API routing.

Zuul is an API gateway for microservices

The red circle is an API gateway ??

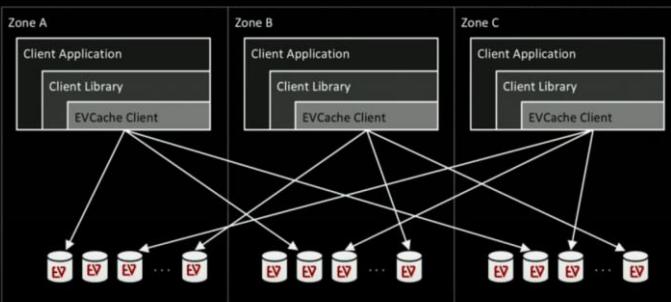
Microservices are an abstraction



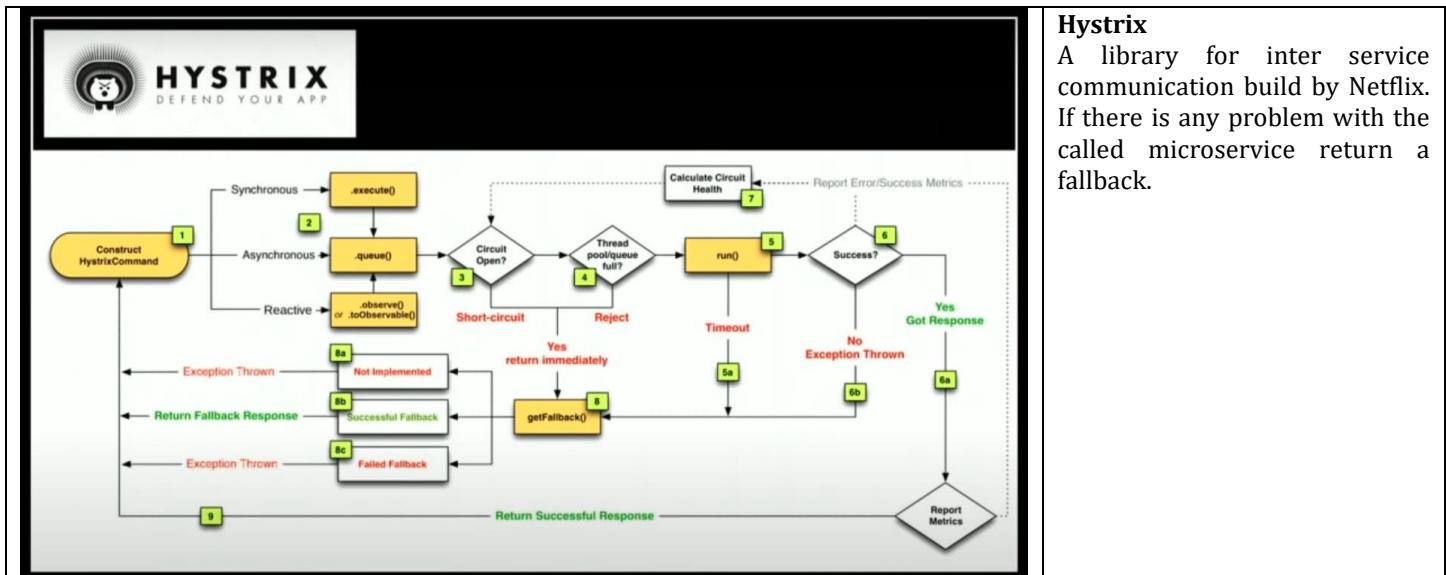
Client library vs bare bone REST

There are pros and cons. They finally chose client libraries. For each microservice they built client library for calling it. the client application (might be a client device or another microservice) uses this client library.

EVCache Writes



Evcache is a lib they wrote on top of memcached. It writes cache data to multiple nodes across availability zones (to avoid single point of failures for cache nodes). it reads from the local nodes for decreased latency but falls back to cross zone reading.



Misc

Patterns for microservices

configuration management

service discovery (eureka library)

circuit breakers

intelligent routing

micro-proxy

control bus

one-time tokens

global locks

leadership election

distributed sessions

cluster state

Spring cloud (spring is a java application framework) is a collection of libraries for implementing common patterns in microservice architectures.

PACT

It is a tool for contract testing. Contract is the external endpoints a microservice provides to the world. You can change its internals but not this contract if you want other systems that interact with you not break. Pact is a tool for testing if this contract has changed.

Deploying Microservices

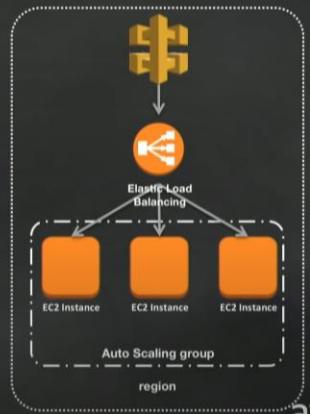
Hystrix

A library for inter service communication build by Netflix. If there is any problem with the called microservice return a fallback.

Deploying Microservices on Amazon EC2

Recommendation:

- Single service per host
- Start with small instance sizes
- Leverage Auto Scaling and AWS Elastic Load Balancing/Application Load Balancer/Network Load Balancer(if in VPC)
- Automate the ability to pump out these environments easily
 - Leverage CodeDeploy, CloudFormation, Elastic Beanstalk or Opsworks



Deploying Microservices with ECS

Recommendation

- Put multiple services per host
- Make use of larger hosts with much more CPU/RAM
- Run helper services on the same host as other dependent services
- Leverage Auto Scaling and AWS Elastic Load Balancing/Application Load Balancer/Network Load Balancer(if in VPC)
- Use AWS Fargate for even less administrative overhead!

