Do One Thing Well

Nathan Taylor

SOFTWARE ENGINEER

@taylonr taylonr.com

The Unix Philosophy

Unix Philosophy

Make each program do one thing well

Expect the output of every program to become the input of another

Design software to be tried early

Use tools over unskilled labor

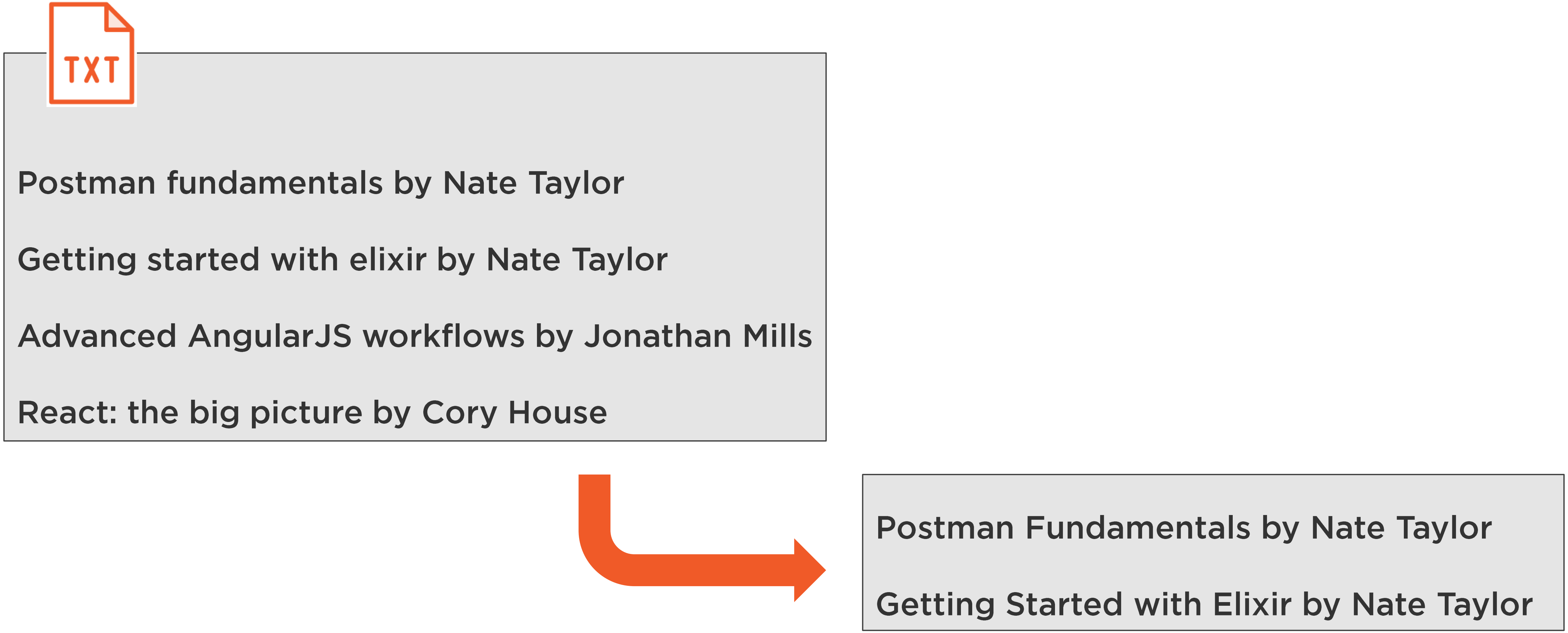
Functional Programming Philosophy

Make each function do one thing well

Expect the output of every function to become the input of another

Design functions to be tested early

Focus on One Thing



Postman fundamentals by Nate Taylor

Getting started with elixir by Nate Taylor

Advanced AngularJS workflows by Jonathan Mills

React: the big picture by Cory House

Postman Fundamentals by Nate Taylor

Getting Started with Elixir by Nate Taylor

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

```
public string[] FindCoursesForAuthor(string author)
{
    var courseFile = File.read(FileName);
    var courses = courseFile.split(\n);
    var authorCourses = [];

    for(var i = 0; i < courses.length; i++)
    {
        if(courses[i].Contains(author))
        {
            authorCourses.push(courses[i].ToTitleCase());
        }
    }

    return authorCourses;
}
```

Function Responsibilities

Opening a file

Splitting the contents

Filtering the courses

Title casing strings

A Functional Example

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

A Functional Example

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

That’s cheating!

Do **one** thing well

Type Signatures

Data Types

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

Data Types

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

Data Types

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

C# Generics

```
public IEnumerable<T> FindCoursesForAuthor<T> (T author,
IEnumerable<T> courses)
```

C# Generics

```
public IEnumerable<T> FindCoursesForAuthor<T> (T author,
IEnumerable<T> courses)
```

Type Signature

```
findAuthorCourses :: (String, [String]) -> [String]
```

Type Signature

```
findAuthorCourses :: (String, [String]) -> [String]
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

Type Signature Explained

```
findAuthorCourses :: (String, [String]) -> [String]
```

Type Signature Explained

```
findAuthorCourses :: (String, [String]) -> [String]
```

Type Signature Explained

```
findAuthorCourses :: (String, [String]) -> [String]
```

Generic Type Signature

```
findAuthor :: (a, [a]) -> [a]
def findAuthor author, authors
    filter(x => x.contains(author), authors)
end
```

Second Type Signature

```
(a, [b]) -> [a]
```

Ask “Can I genericize this?”

Why Type Signatures?

Prevalent in documentation

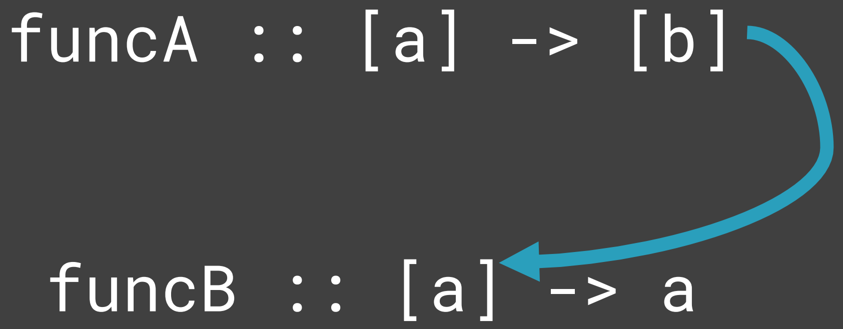
100 functions on 1 data structure

Simplify second principle

From Output to Input

Type signatures show how functions can be chained

Example Type Signatures



```
funcA :: [a] -> [b]

funcB :: [a] -> a
```

Reusing Output

```
var tempResult = funcA([{name: 'Nate Taylor', location:
'Omaha'}]);

//tempResult = ['Omaha']


var finalResult = funcB(tempResult);
```

Reusing Output

```
var finalResult = funcB(funcA([{name: 'Nate Taylor',
location: 'Omaha'}]));
```

|>

Pipe operator

Example Type Signatures

```
funcA([{name: 'Nate Taylor', location: 'Omaha'}])
|> funcB()
```

Original Function

```
def findAuthorCourses author, courses
    filter(x => x.contains(author), courses)
end
```

Function Chain

```
loadFile('courses')
|> splitOnNewLine()
|> findAuthorCourses(author)
|> titleCase()
```

Function Chain

```
loadFile('courses')
|> splitOnNewLine()
|> findAuthorCourses(author)
|> titleCase()
```

Function Chain

```
loadFile('courses')
|> splitOnNewLine()
|> findAuthorCourses(author)
|> titleCase()
```

Function Chain

```
loadFile('courses')
|> splitOnNewLine()
|> findAuthorCourses(author)
|> titleCase()
```

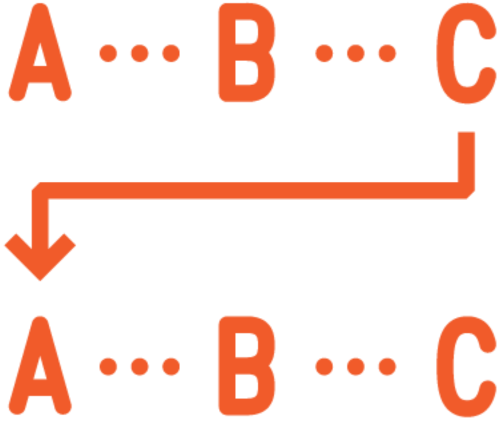
Function Chain

~~|> findAuthorCourses(author)~~

```
findAuthorCourses(author, courses)
```

Function Chain

```
loadFile('courses')
|> splitOnNewLine()
|> findAuthorCourses(author)
|> titleCase()
```



Chaining allows easier decomposition

Focused functions lead to higher reuse

Test Early

Always test early

How early is early?

Write unit tests

Functional Programming Advantages

Easier set up

Easier verification

Testing functional programming is straight forward

Complexity through Simplicity

Approach to Functional Programming

Decompose problems

Reuse building blocks

Test early

Learning functional programming involves a shift in perspective