AWS CLOUD FORMATION

Concise developer – Alex HD

Short definition: Infrastructure as code solution provided by AWS.

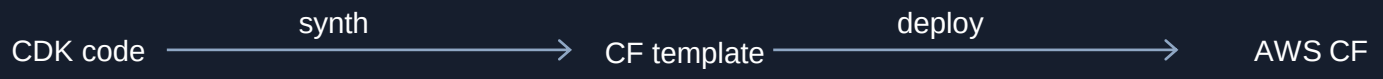
Short definition: Infrastructure as code solution provided by AWS. CF Stack: multiple AWS resources handled together

Template file (JSON, YAML) → Stack

With functions:

- get
- ref
- import

…



AWS CDK Constructs

L1

Low level constructs – Cfn(Cloud formation) resources. When used, we must configure all properties.

L2

AWS resources with a higher-level – CDK provides additional functionality like defaults, boiler plate and type safety for many parameters

L3

Patterns: Combine multiple types of resources and help with common tasks in AWS.

Examples: LambdaRestApi

When to use with CDK?

L1

Most AWS resources are migrated to L2. Use for new services that are still not migrated

L2

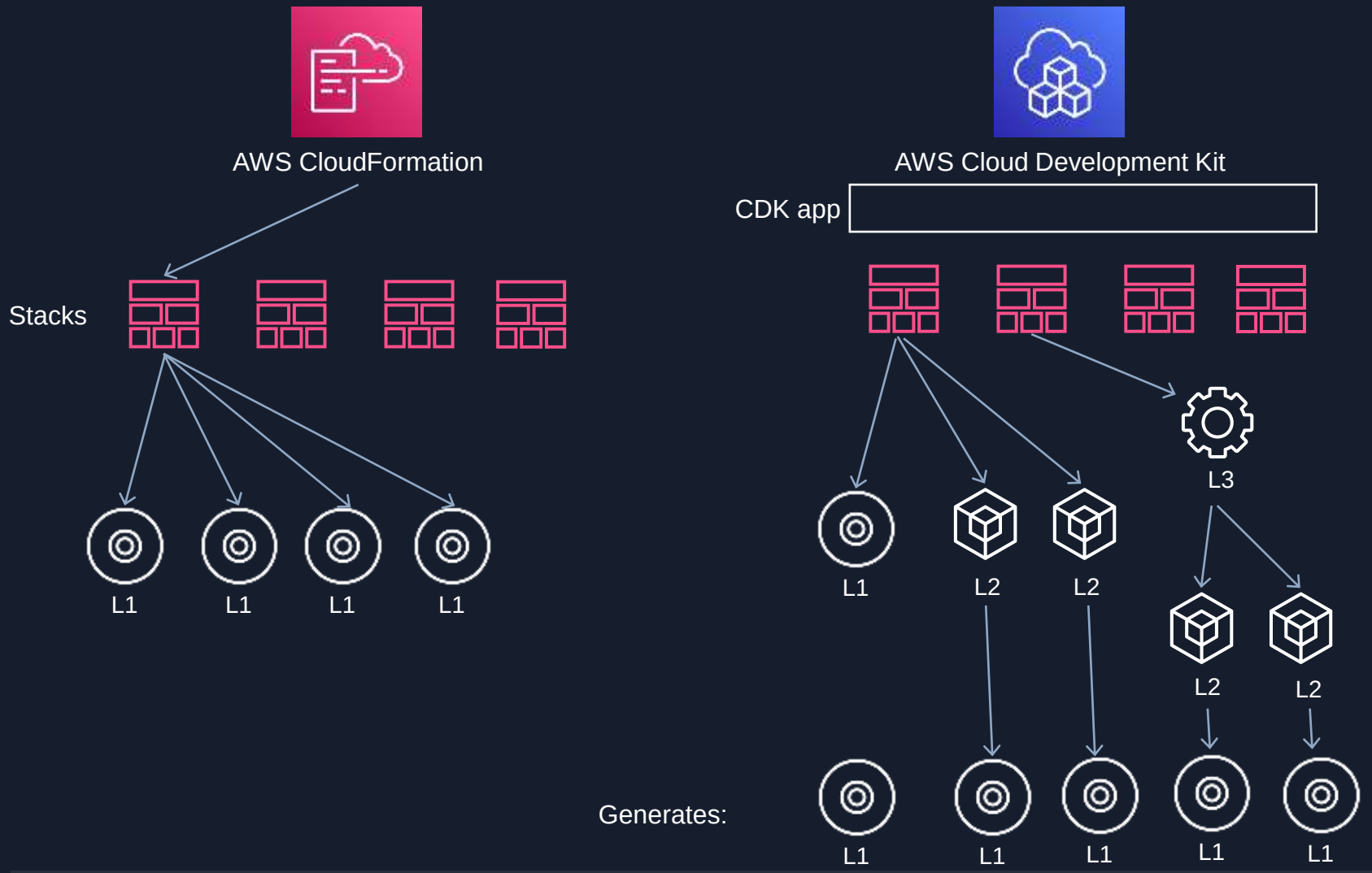
Most of the time

L3

Matter of preference of company policy.

What degree of abstraction do I want?

CloudFormation vs CDK:

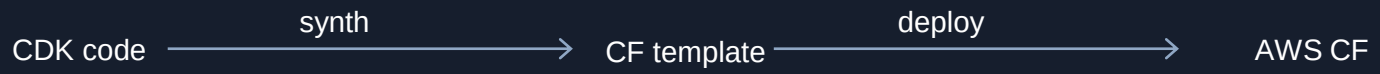


Objective of an abstraction:

Simplify, not complicate!

CLOUD FORMATION INTRINSIC FUNCTIONS

Short definition: build-in functions to help manage our stacks.



ZOOM on deployment step:

Stack is created. Available information:

- stack name, stack id …
- deployment parameters, external parameters

Resource1 is created. Ex: s3 bucket. Available info: bucket name, id …

Resource2 is created. Ex: s3 bucket. Available info: bucket name, id …

…..

Deployment finished

CloudFormation intrinsic functions:

- Fn::Base64
- Fn::Cidr
- Fn::FindInMap
- Fn::GetAtt
- Fn::GetAZs
- Fn::ImportValue
- Fn::Join
- Fn::Length
- Fn::Select
- Fn::Split
- Fn::Sub
- Fn::ToJsonString
- Fn::Transform
- **Ref**

Condition functions – Fn:If – CDK provides great abstractions

HANDLING MULTIPLE STACKS

Why the need for multiple stacks?

- Some stacks may contain sensitive info(IAM roles, secrets, …)
- Some stacks may take a lot of time for deployment od deletion
- Resources get big and the need organization

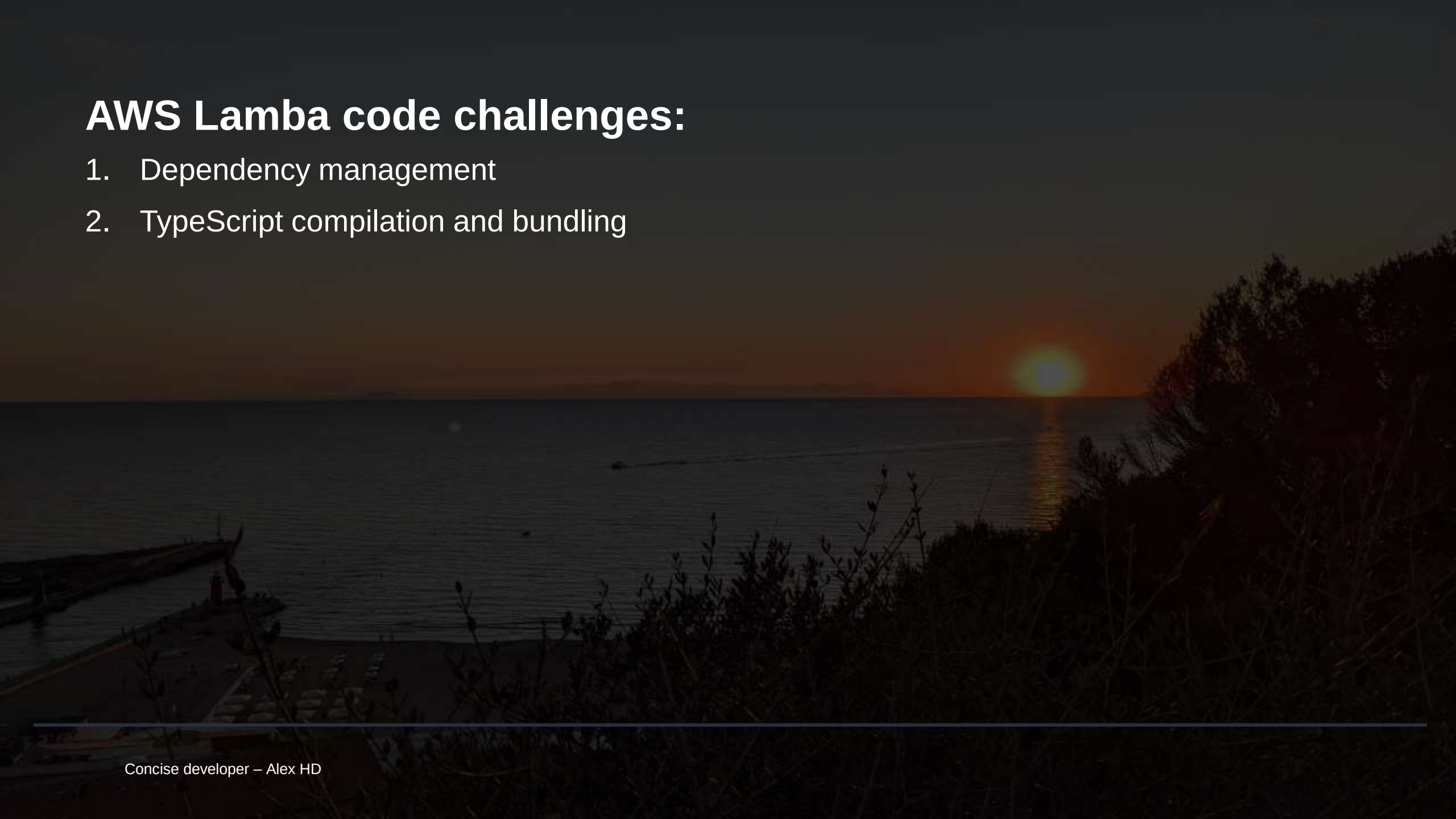
How to organize stacks?

- There are no documented rules, not even best practices
  - Separate stacks for resources with state(databases, buckets)

  - Separate stacks for IAM roles, policies, secrets

  - Separate stacks for resources with complex initialization(VPCs, DNS)

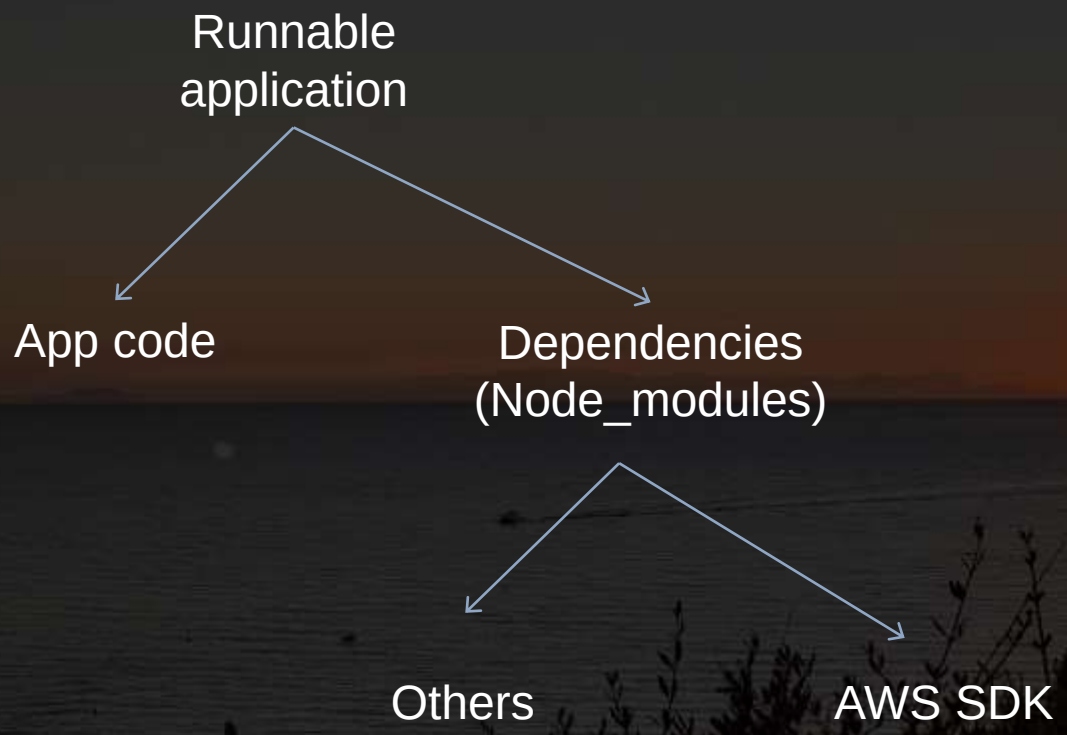
Challenge: cross stack references

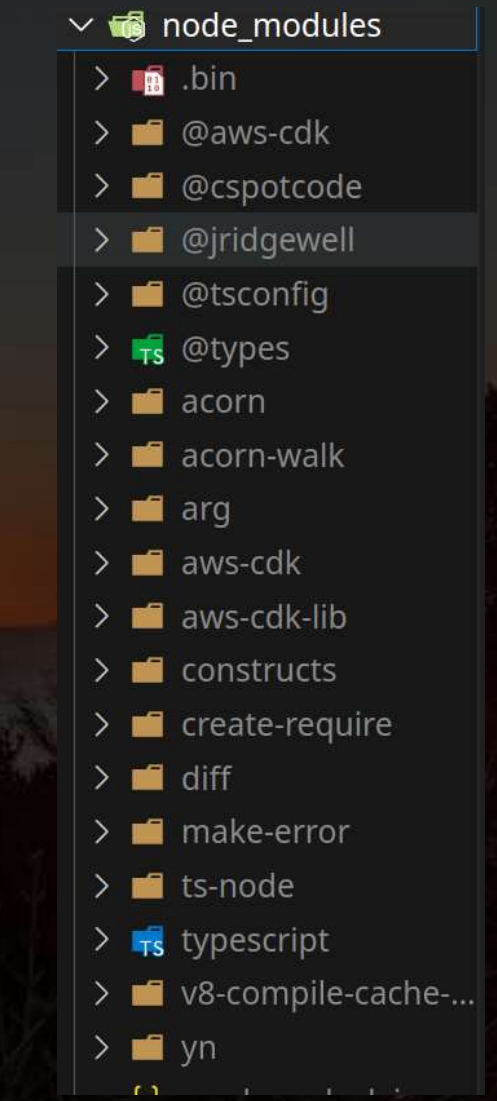


AWS Lamba code challenges:

1. Dependency management
2. TypeScript compilation and bundling

Dependency management:

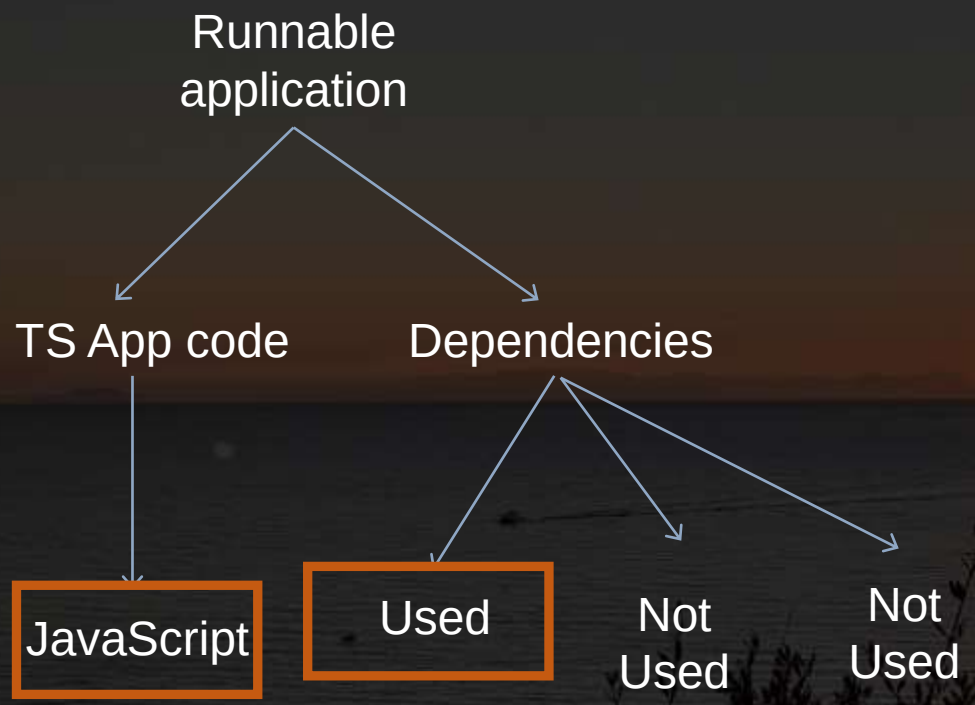


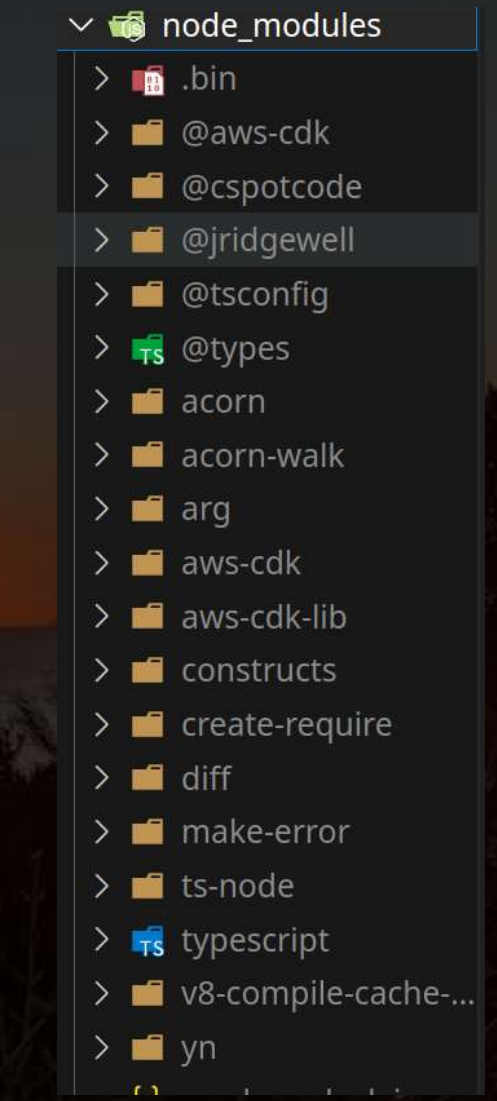


Dependency management :

1. Deploy only dependencies, not dev dependencies(TS, TS-node, CDK, etc …)
2. Do not deploy AWS SDK dependencies – included in the Lambda runtime

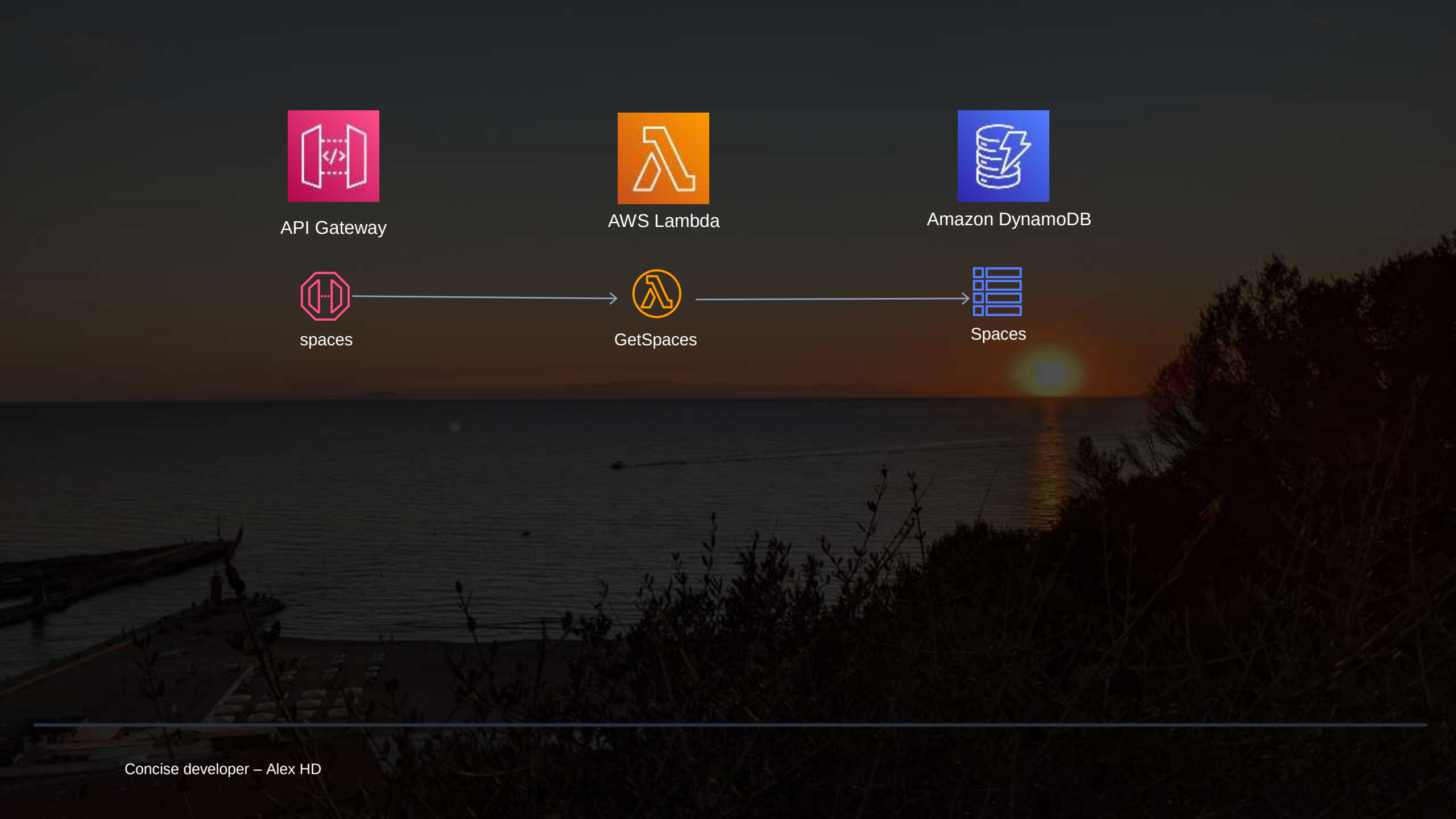
TypeScript compilation and bundling :





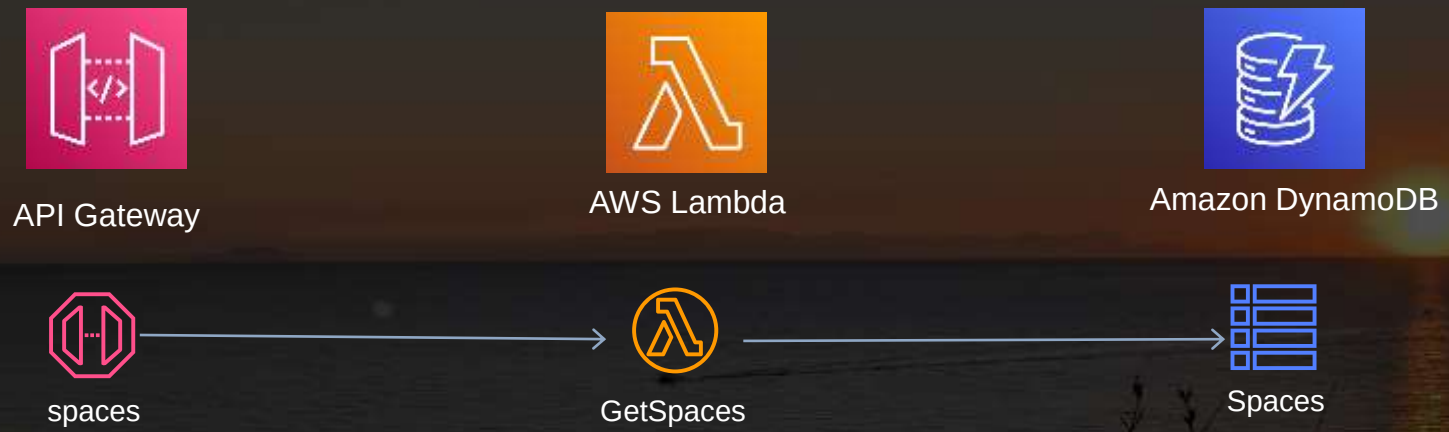
Solution: NodejsFunction CDK construct

1. Bundles all code with three shaking
2. Compiles TS to JS
3. Leaves out AWS-SDK dependencies
4. Completely editable
5. Library: esbuild
    1. Past solution: webpack – slow and hard to configure


API Gateway
AWS Lambda
Amazon DynamoDB
spaces
GetSpaces
Spaces

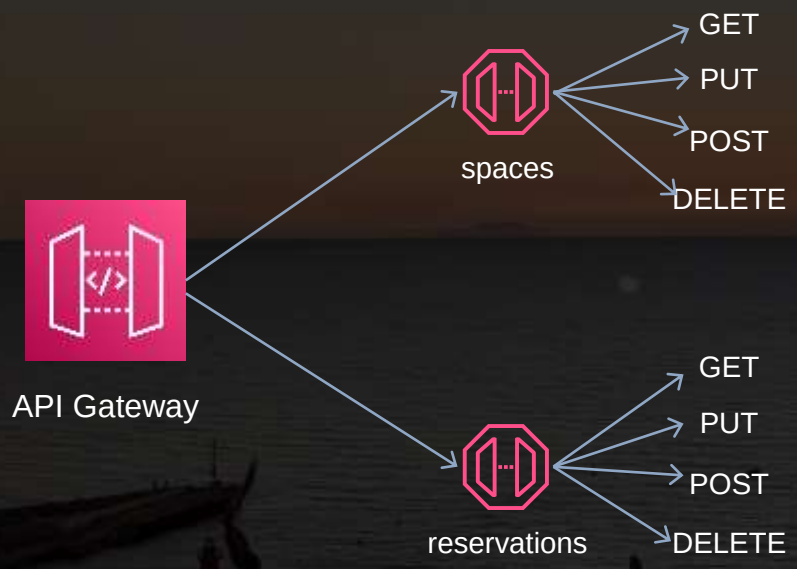
AWS Lambda architecture:

1. Easy for basic apps:



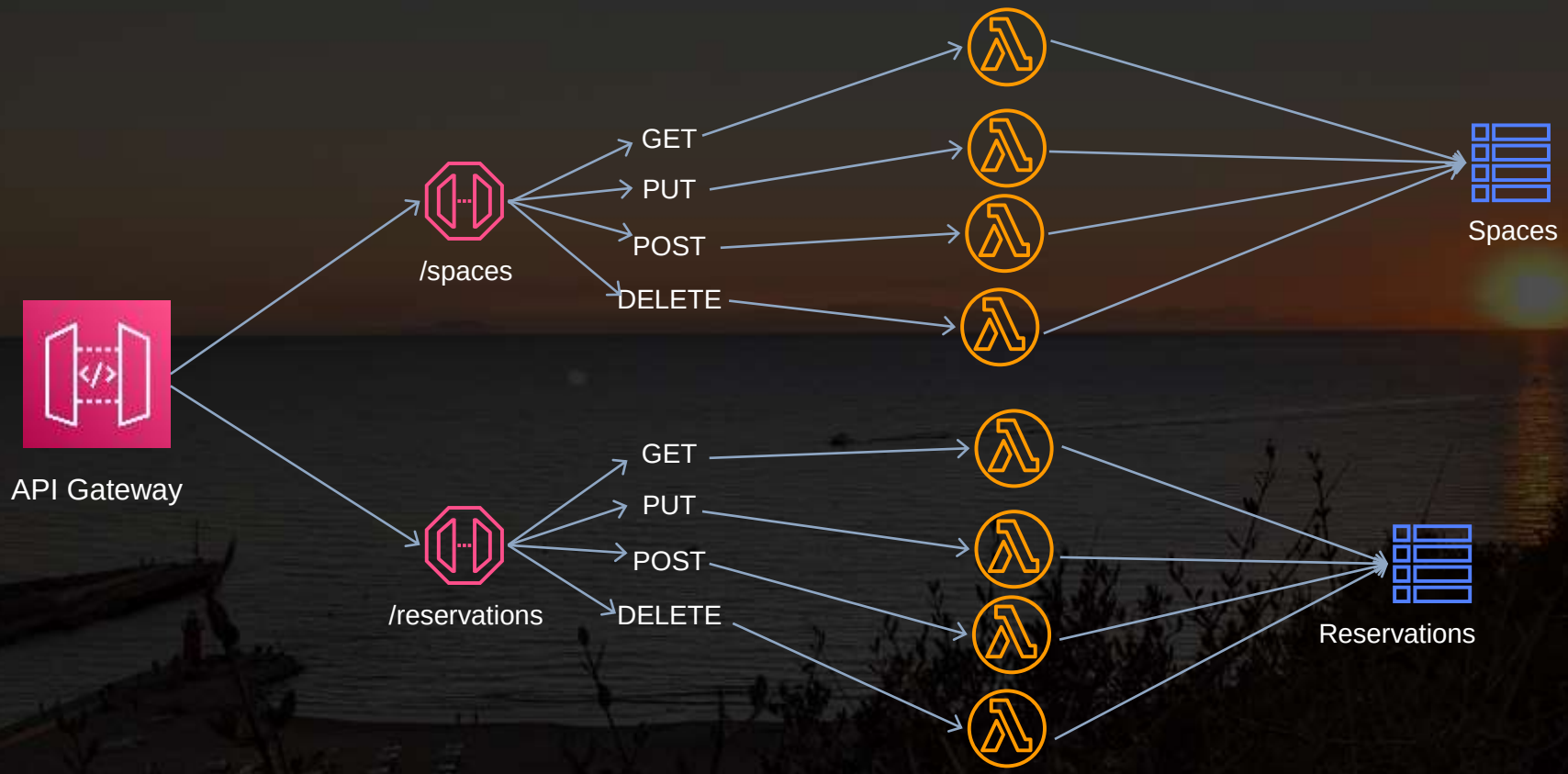
AWS Lambda architecture:

1. How about extending the app?



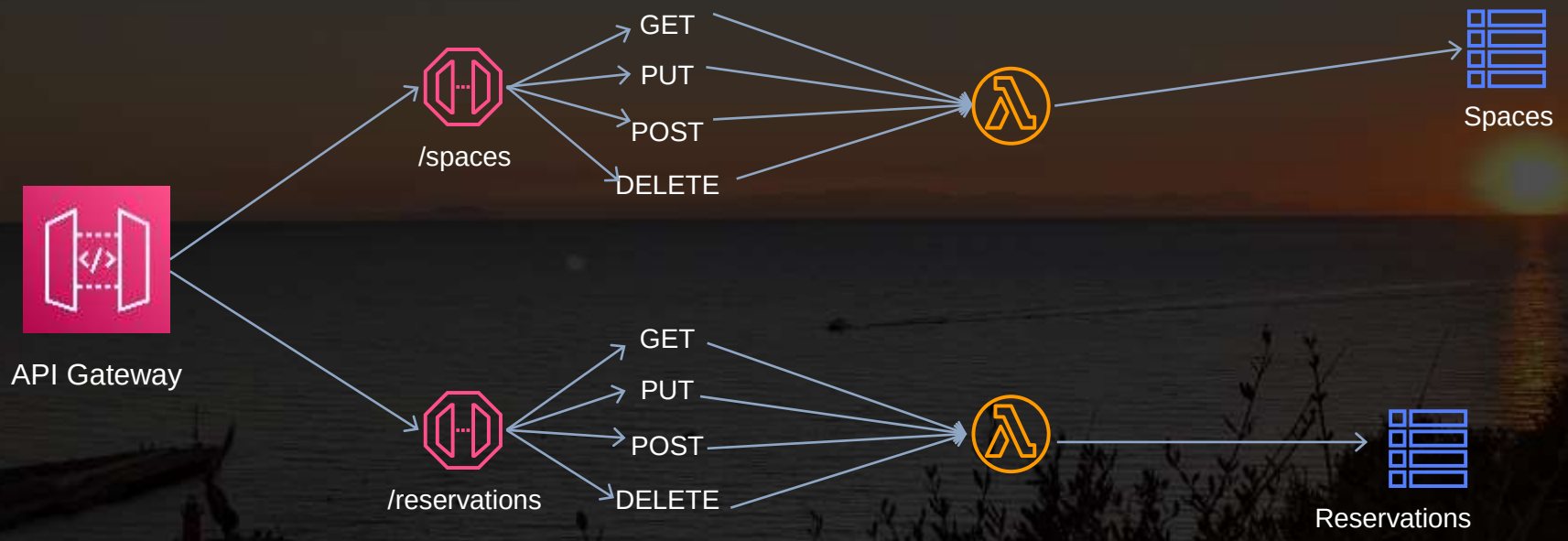
AWS Lambda architecture:

1. Multiple lambdas



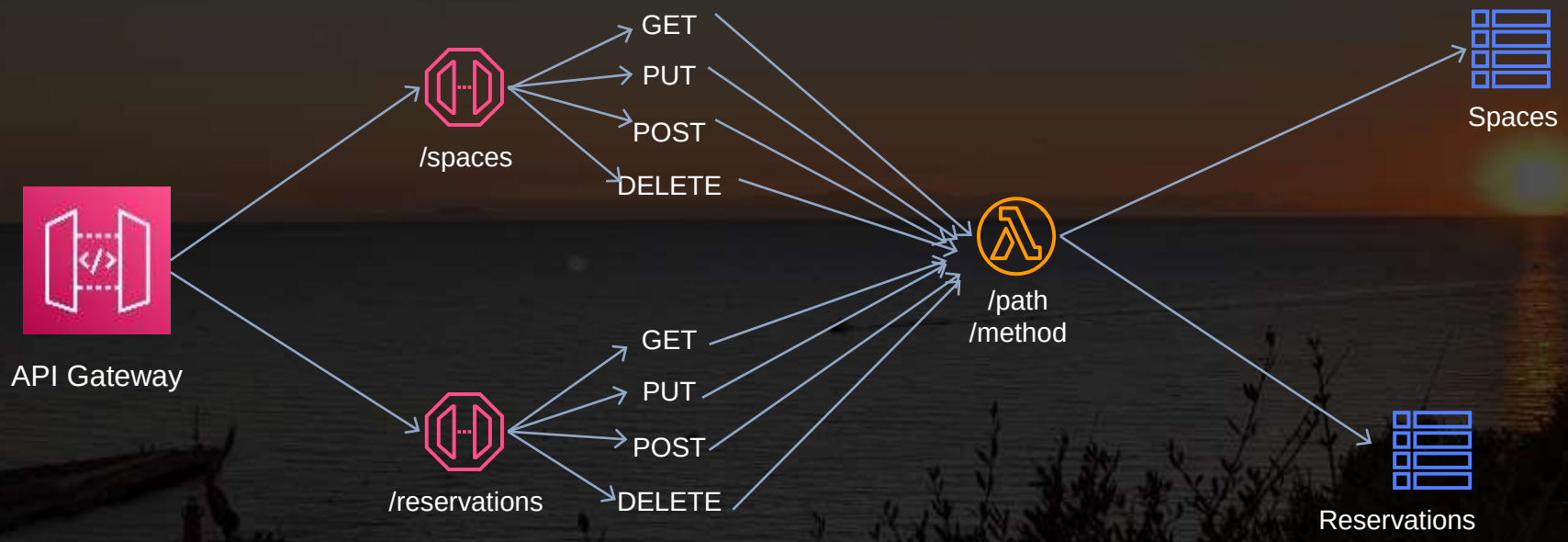
AWS Lambda architecture:

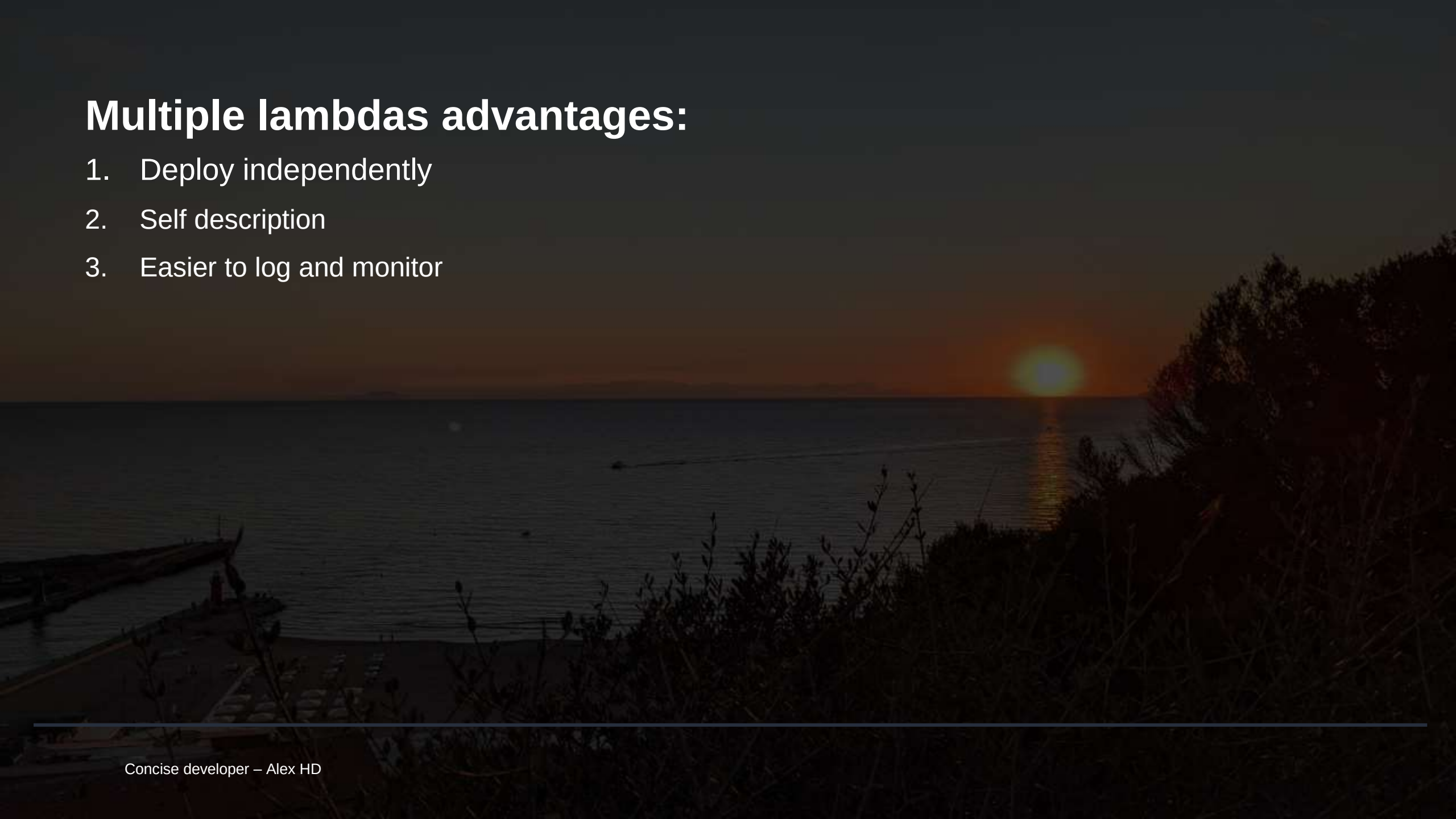
1. Group by API Gateway resource



AWS Lambda architecture:

1. Monolithic lambda



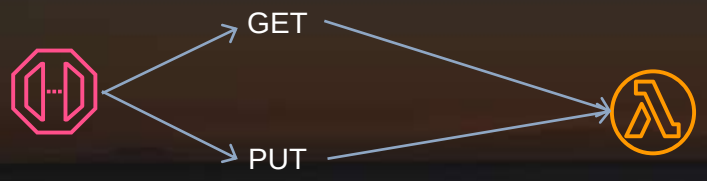


Multiple lambdas advantages:

1. Deploy independently
2. Self description
3. Easier to log and monitor

Let’s build now:

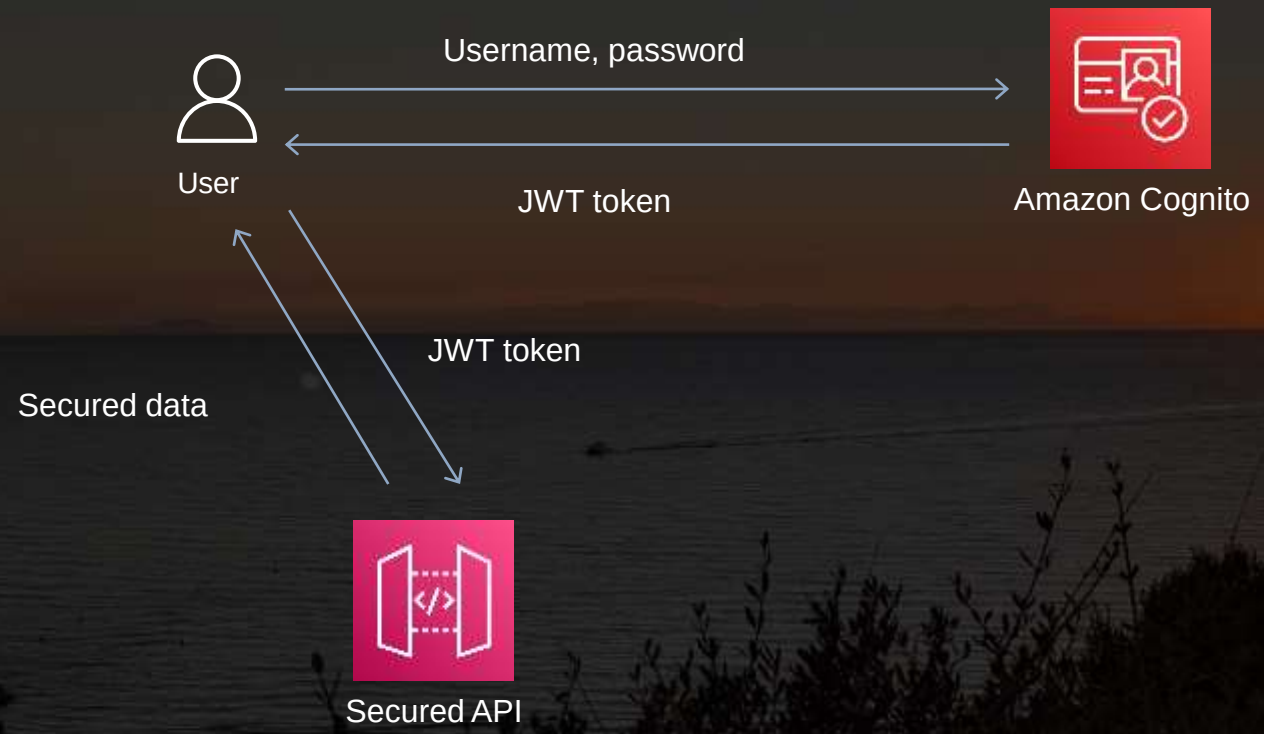
1. ApiGateway handles routing
2. Lambda handles HTTP method



Understanding AWS COGNITO

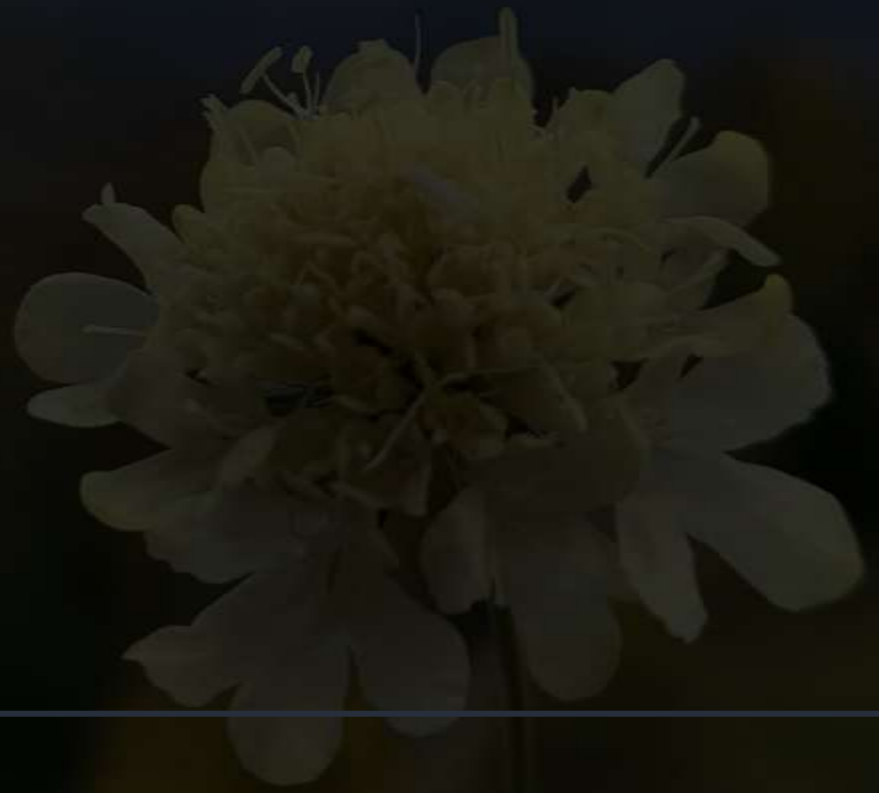
1. User pools
    1. Stores user data
    2. Basic authentication solution – JWT tokens
2. Identity pools
    1. Fine grained access control – a user assumes an identity
    2. Can directly call AWS SDK commands

How user pools work?



Understanding React:

1. Components
2. State/Props
3. Hooks



Understanding React Components:

Components are independent and reusable bits of code:

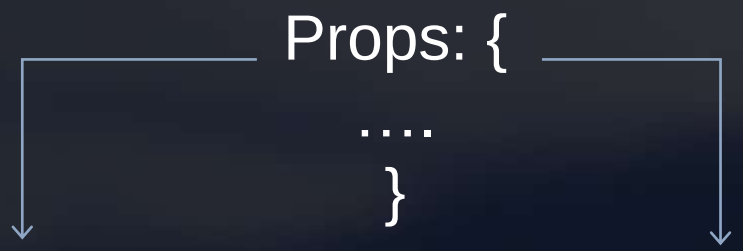
- definition: basic JavaScript function
- inside jsx/tsx files – they can contain both logic and UI
- they return html code with some React extras

```tsx
interface SpaceComponentProps extends SpaceEntry {
  reserveSpace: (spaceId: string, spaceName: string) => void;
}

export default function SpaceComponent(props: SpaceComponentProps) {
  function renderImage() {
    if (props.photoUrl) {
      return <img src={props.photoUrl}/>;
    } else {
      return <img src={genericImage}/>;
    }
  }

  return (
    <div className="spaceComponent">
      {renderImage()}
      <label className="name">{props.name}</label>
      <br />
      <label className="location">{props.location}</label>
      <br />
      <button onClick={() => props.reserveSpace(props.id, props.name)}>Reserve</button>
    </div>
  );
}
```

Understanding React Components – state/props:



State changes => component “reacts” and it is re-rendered

State: {
…
}

Parent: props/state

Child: state/props

Understanding React hooks:

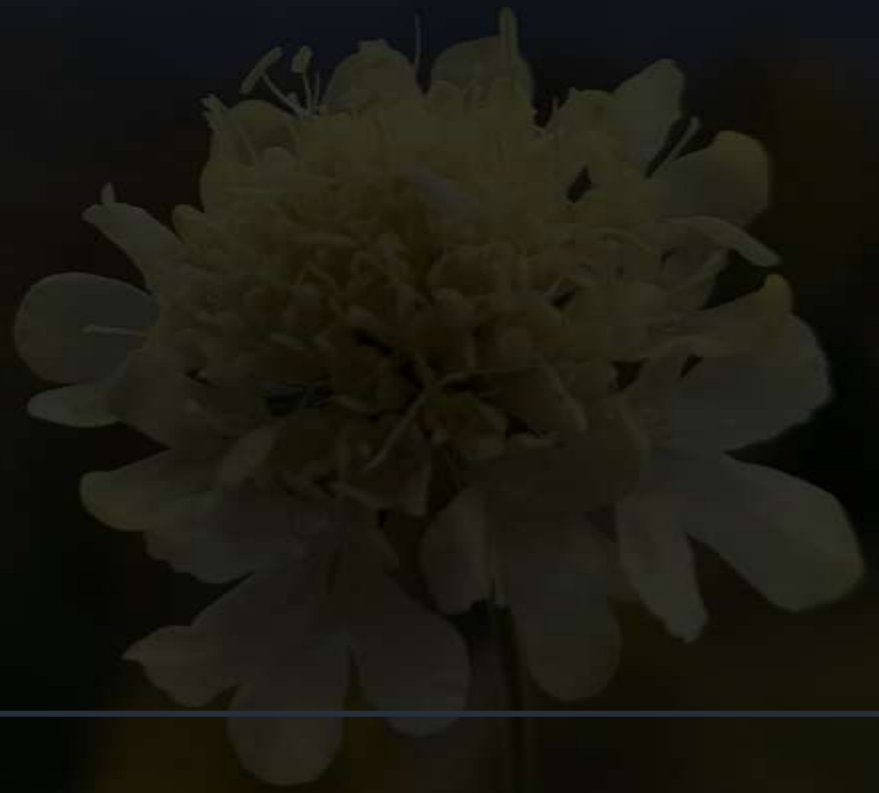
Functions that let us “Hook” inside a component:

- useState – most popular
- useEffect

```
export default function LoginComponent({ authService, setUserNameCb }: LoginProps) {
  const [userName, setUserName] = useState<string>("");
  const [password, setPassword] = useState<string>("");
  const [errorMessage, setErrorMessage] = useState<string>("");
  const [loginSuccess, setLoginSuccess] = useState<boolean>(false);
```

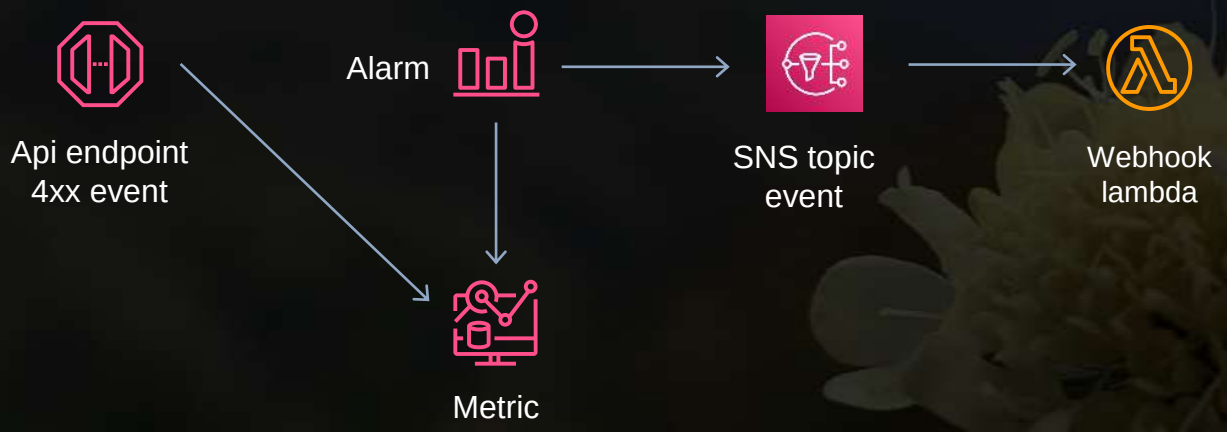
React rant:

It’s like fashion, it’s always evolving



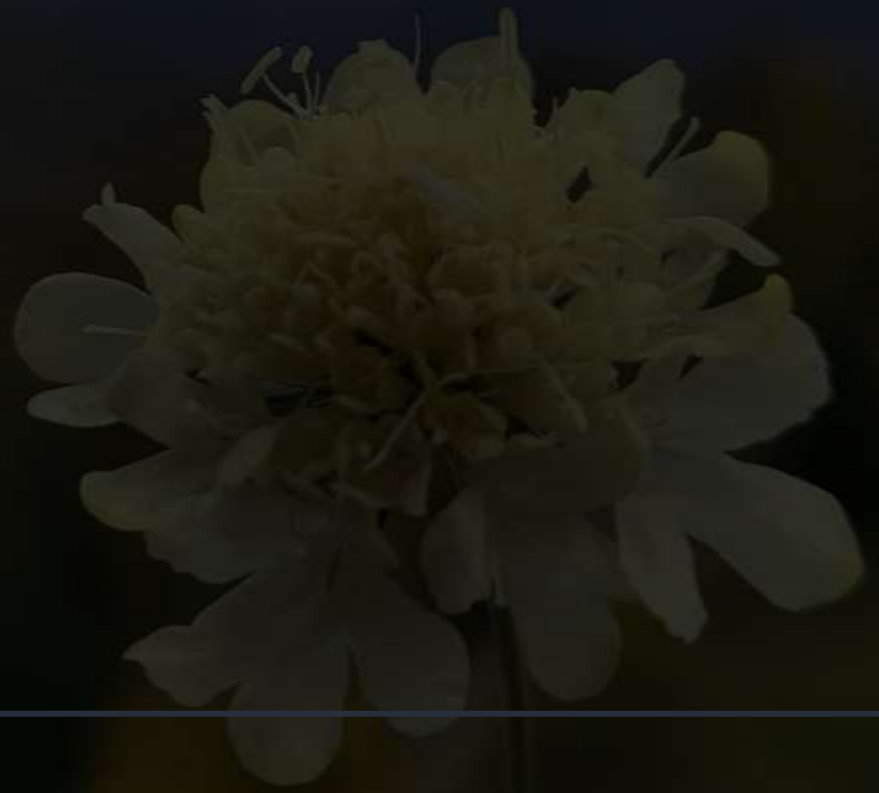
What we will build: Monitoring stack

- Alarm for too many 4xx responses in our API



Services we will use:

- CloudWatch: metrics and alarms
- SNS – handles CloudWatch events and triggers Lambda
- Lambda – makes an http call (to a webhook)



AWS SNS:

- Amazon Simple Notification Service – used for event driven apps