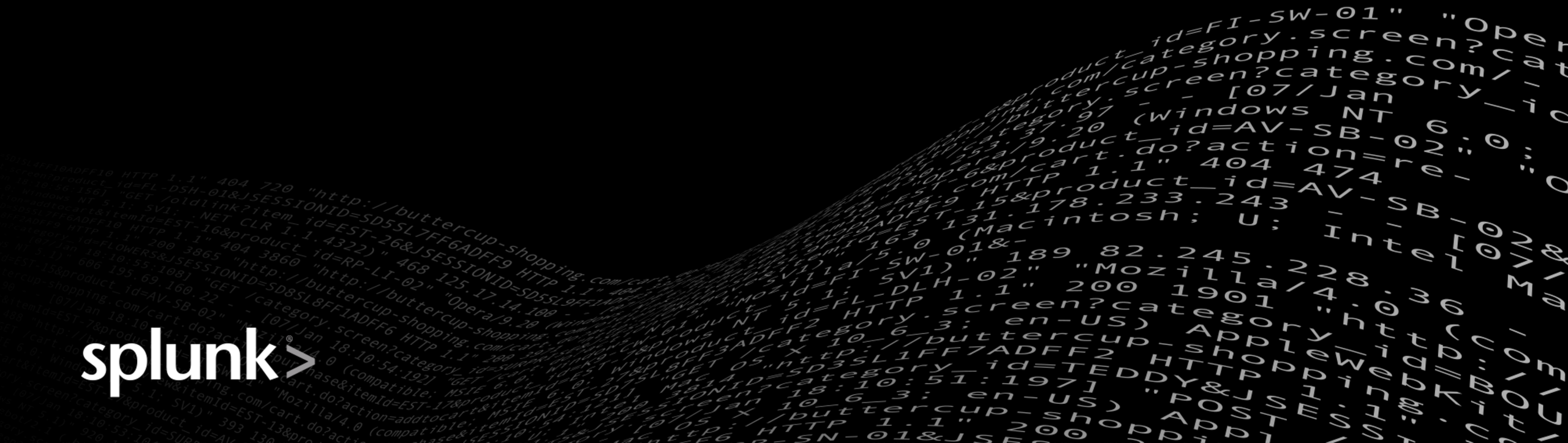
Resource Modeling with REST

Tips for Building a Better REST API

DeAndré Carroll

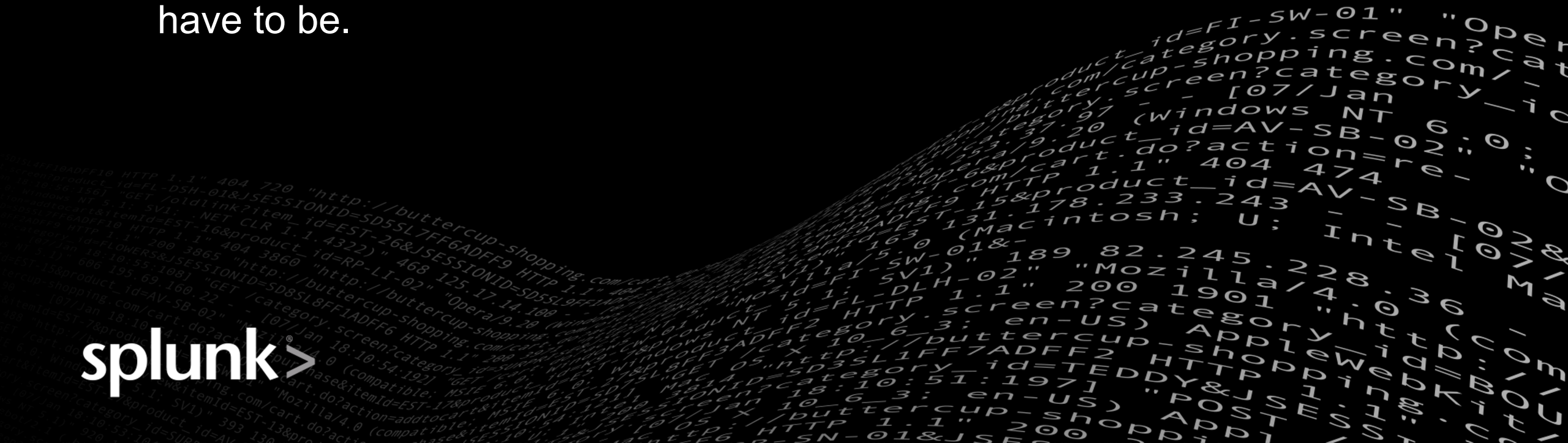
Sr. Software Engineer, Splunk

REST Refresher



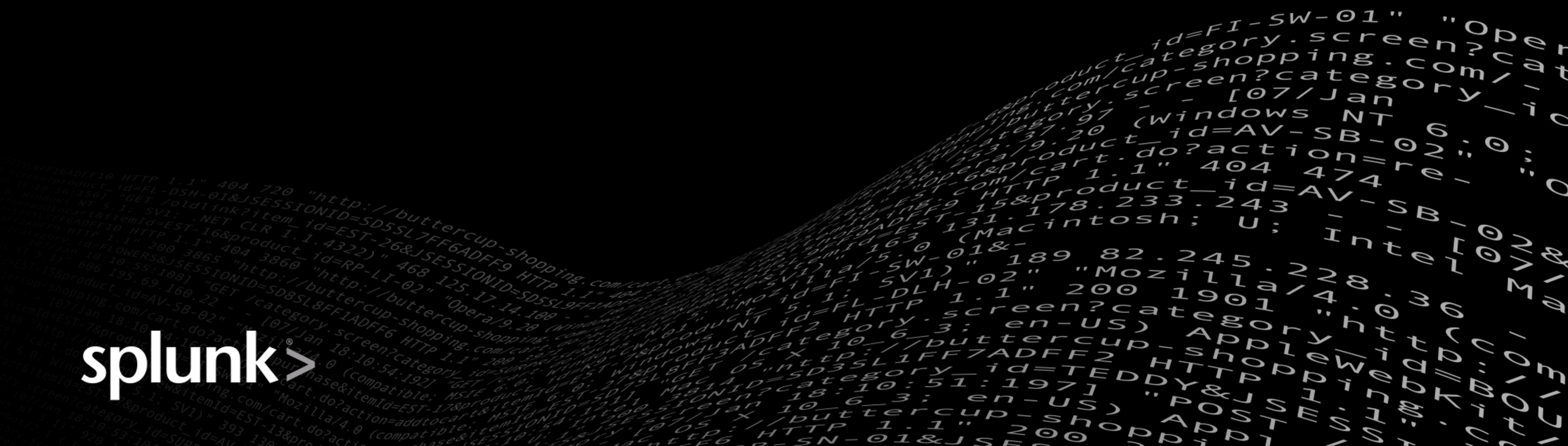
What is REST?

- Representational State Transfer
- Defined by T. Roy Fielding in 2000 for his PhD dissertation.
- Resource oriented architectural style reflected in the structure of the world wide web.
- Usually implemented in HTTP but does not have to be.



Properties of REST?

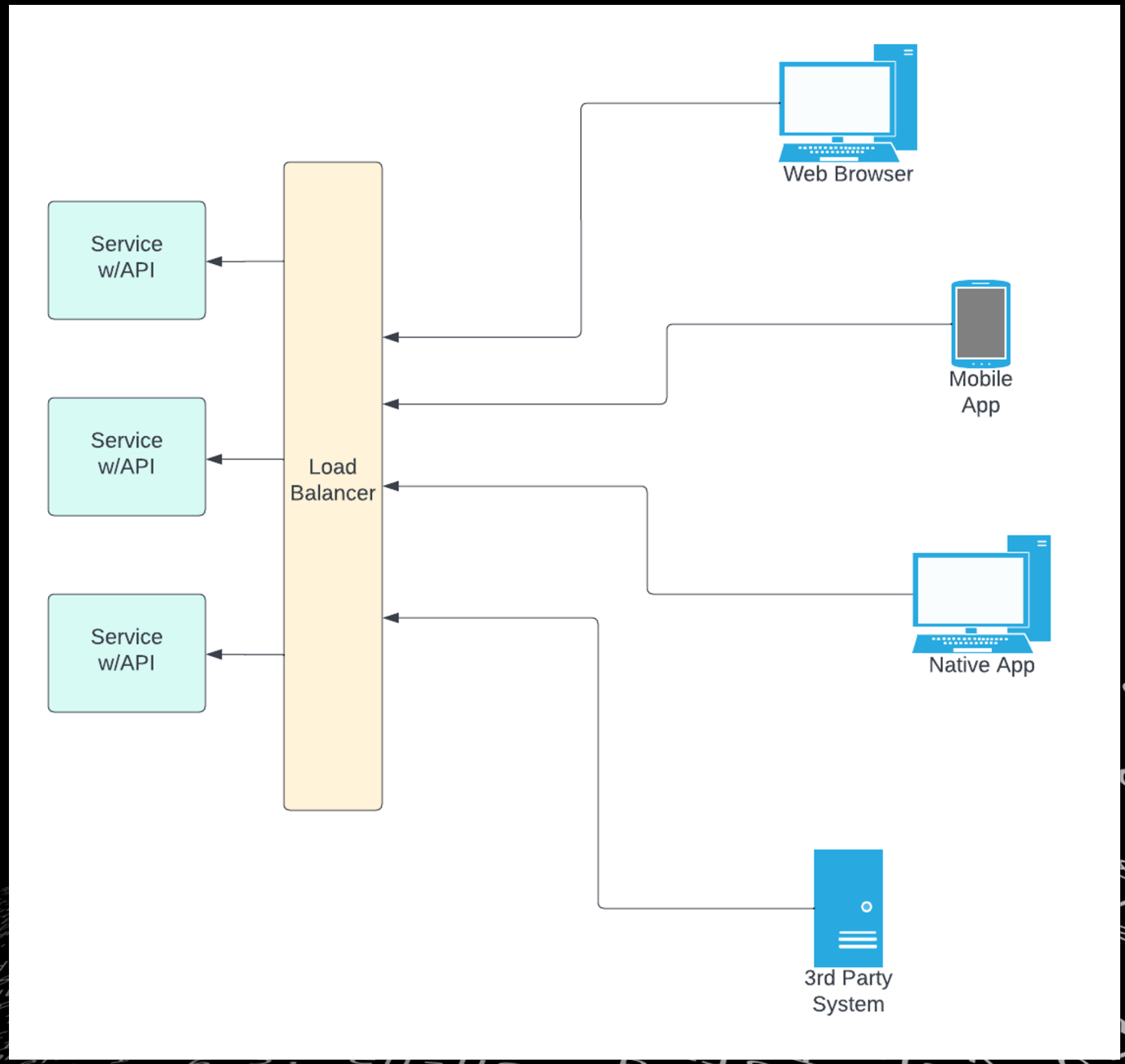
- Client-Server
- Stateless (Communication)
- Cache
- Uniform Interface
- Layered System
- Code-On-Demand



Client-Server

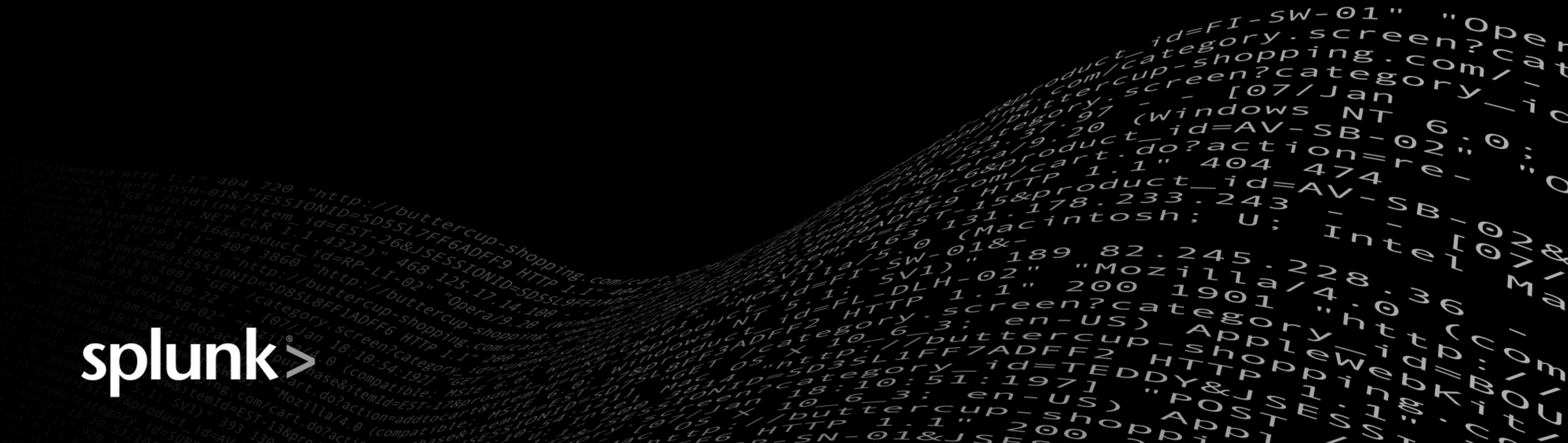
- Separating the user interface concerns from the data storage concerns
  - Improves portability of the user interface
  - Improves scalability by simplifying the server components.

- Separation allows the components to evolve independently



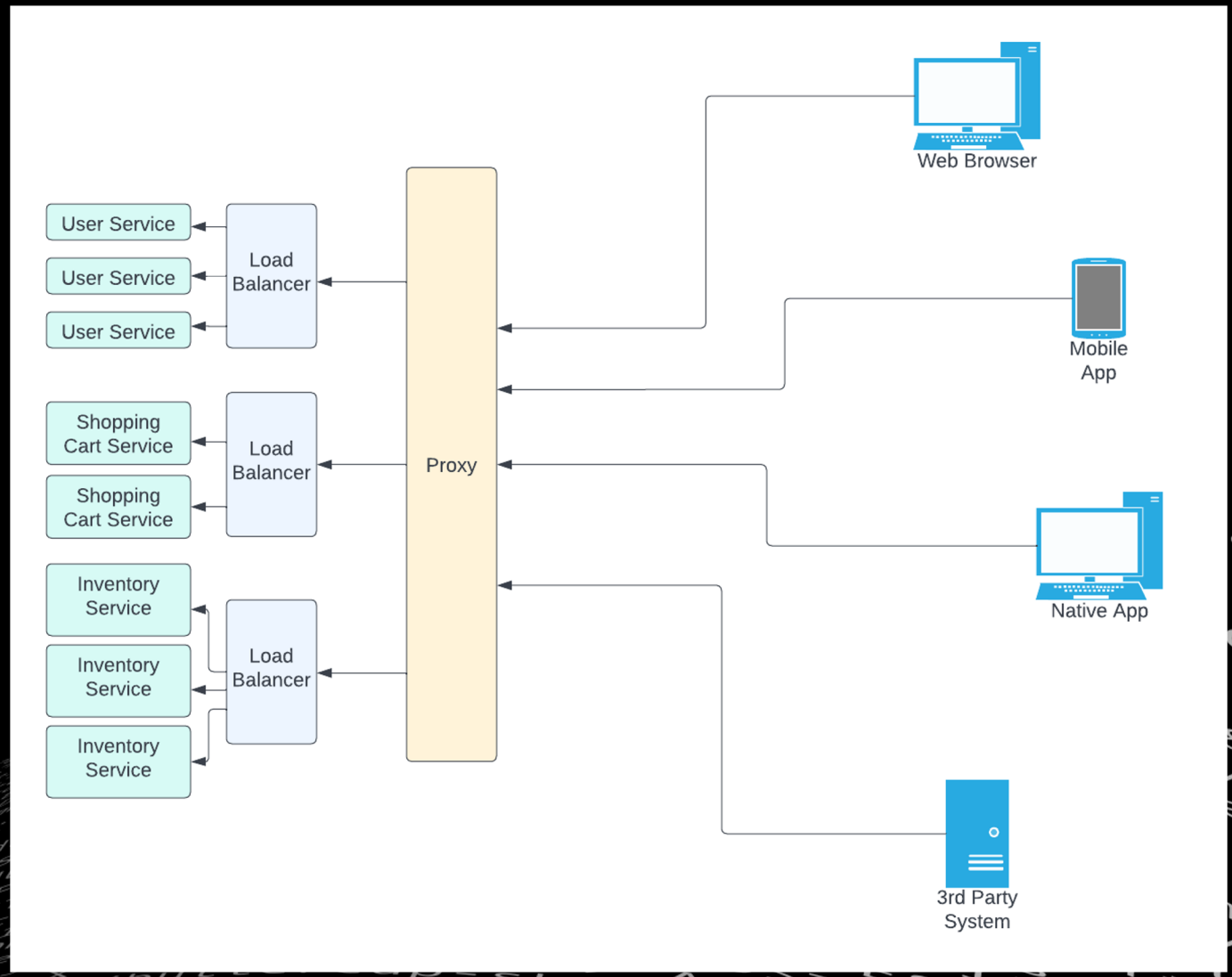
Stateless (Communication)

- Induces the properties of:
  - Visibility
  - Reliability
  - Scalability



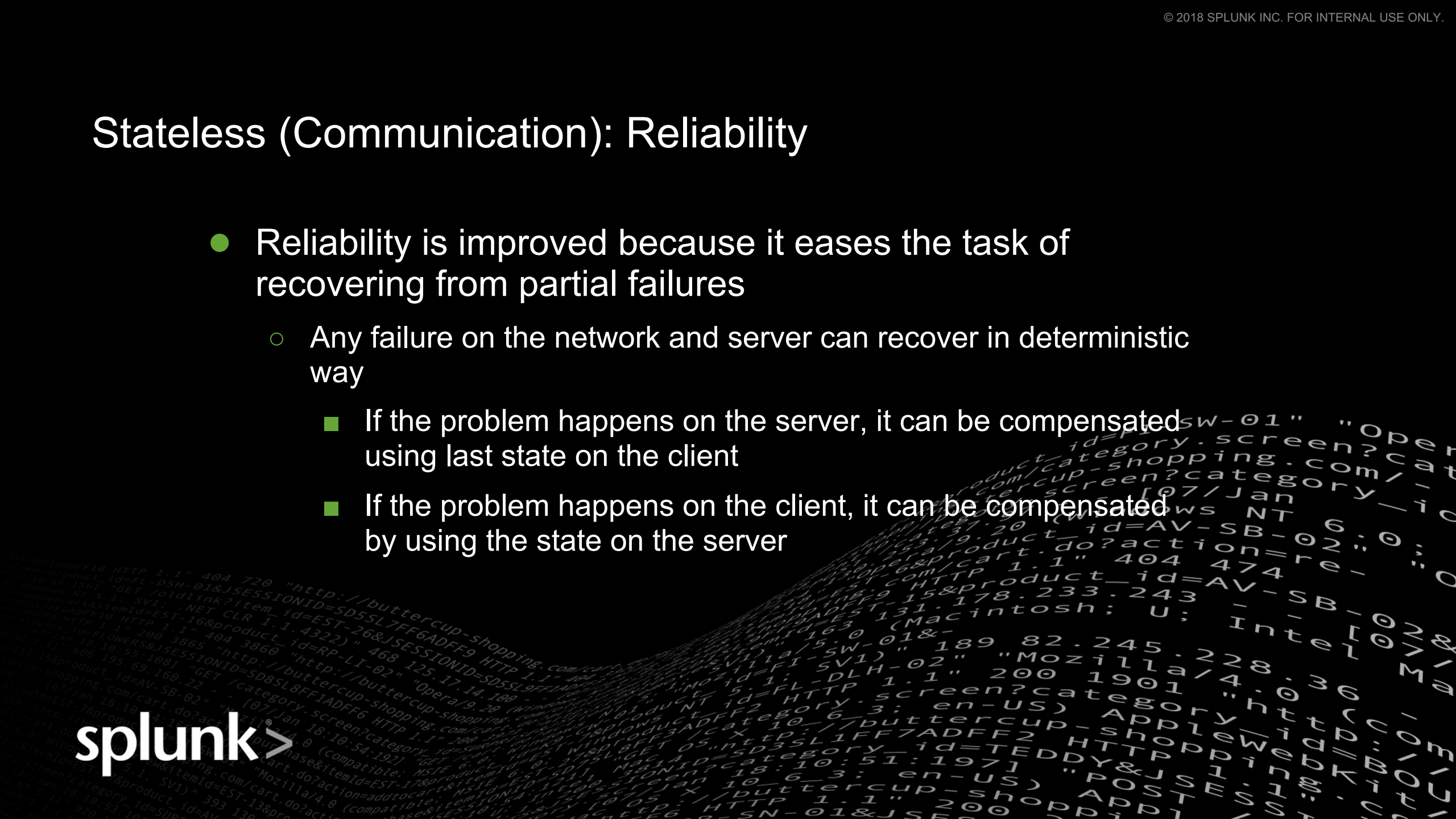
Stateless (Communication): Visibility

- Visibility is “the ability of a component to monitor or mediate the interaction between two other components.”
  - “Visible” because each component of the system can “see” details of the request and resource.
  - Allows additional components (e.g. proxies, caches, etc.) to be added to the system because it isn’t necessary to examine several requests to determine the context of the current request.



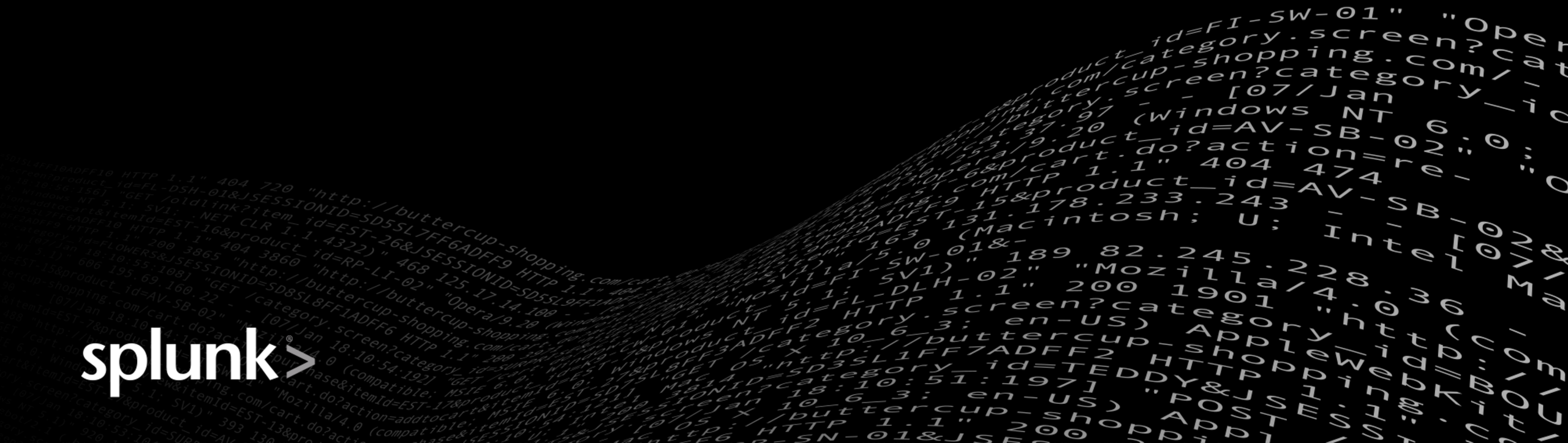
Stateless (Communication): Reliability

- Reliability is improved because it eases the task of recovering from partial failures
  - Any failure on the network and server can recover in deterministic way
    - If the problem happens on the server, it can be compensated using last state on the client
    - If the problem happens on the client, it can be compensated by using the state on the server



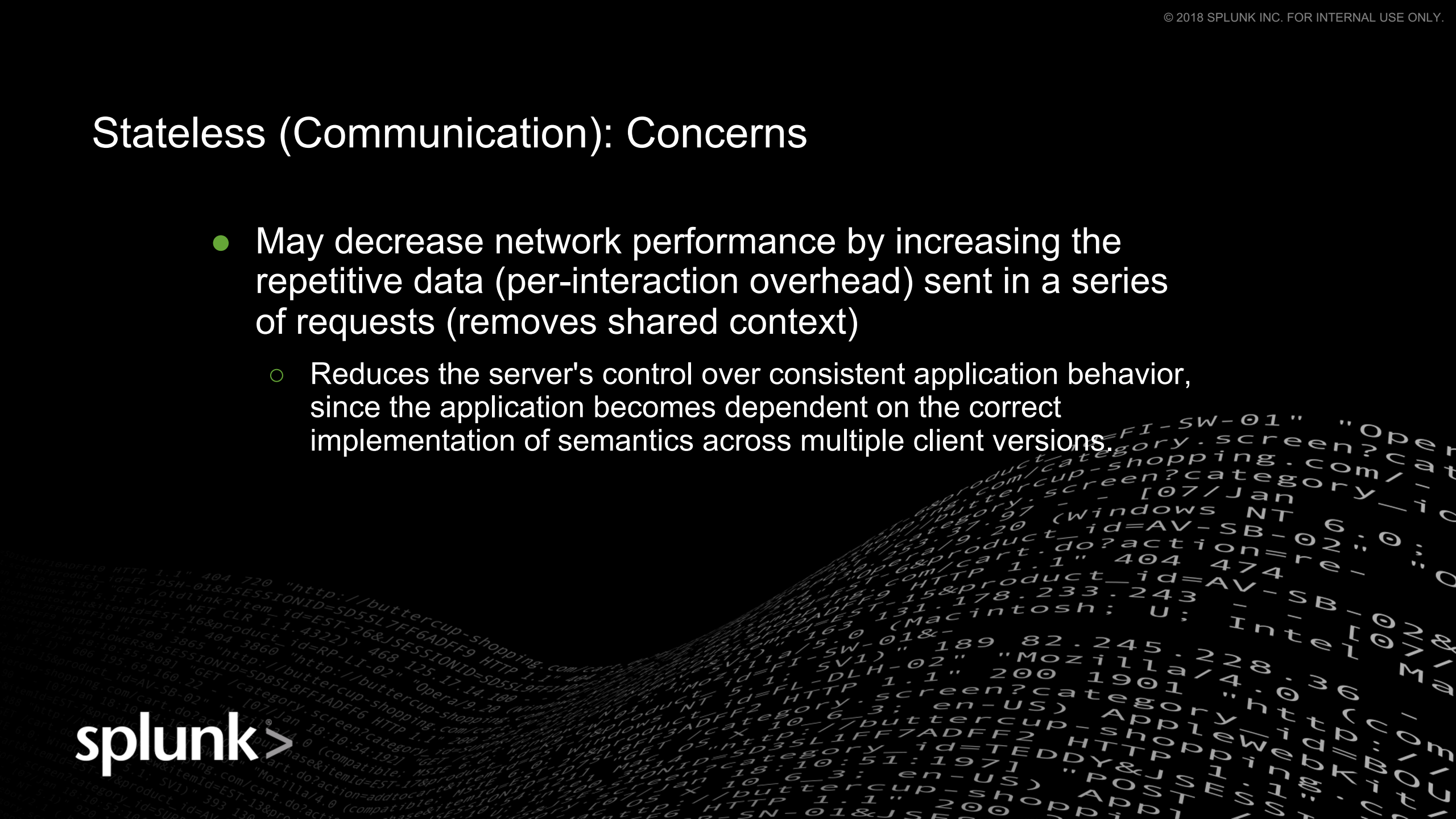
Stateless (Communication): Scalability

- Scalability is improved by allowing the addition of more servers that don’t have to “know” client application state.
  - Application (session) state is therefore kept entirely on the client (No cookies)



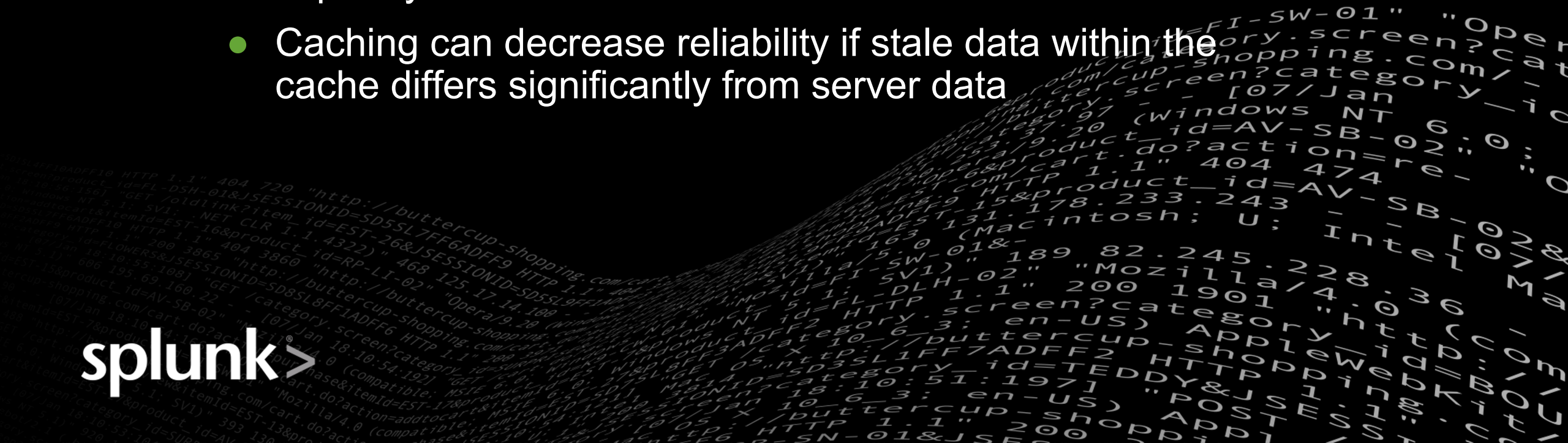
Stateless (Communication): Concerns

- May decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests (removes shared context)
  - Reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of semantics across multiple client versions.



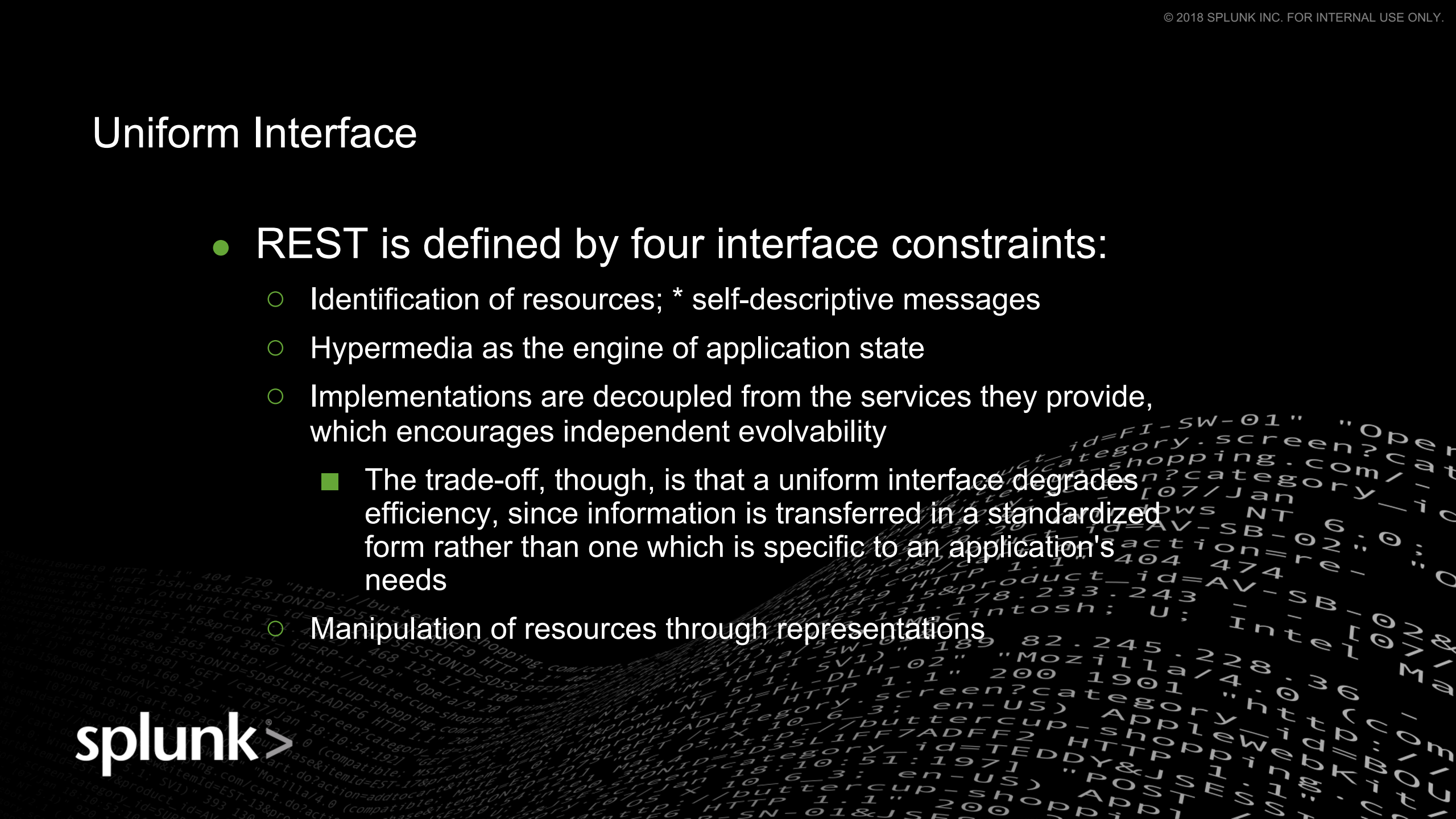
Cache

- Used to improve network performance
- Can be cached on client side or server side
- Data within a response to a request is implicitly or explicitly labeled as cacheable or non-cacheable
- Caching can decrease reliability if stale data within the cache differs significantly from server data



Uniform Interface

- REST is defined by four interface constraints:
  - Identification of resources; * self-descriptive messages
  - Hypermedia as the engine of application state
  - Implementations are decoupled from the services they provide, which encourages independent evolvability
    - The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs
  - Manipulation of resources through representations

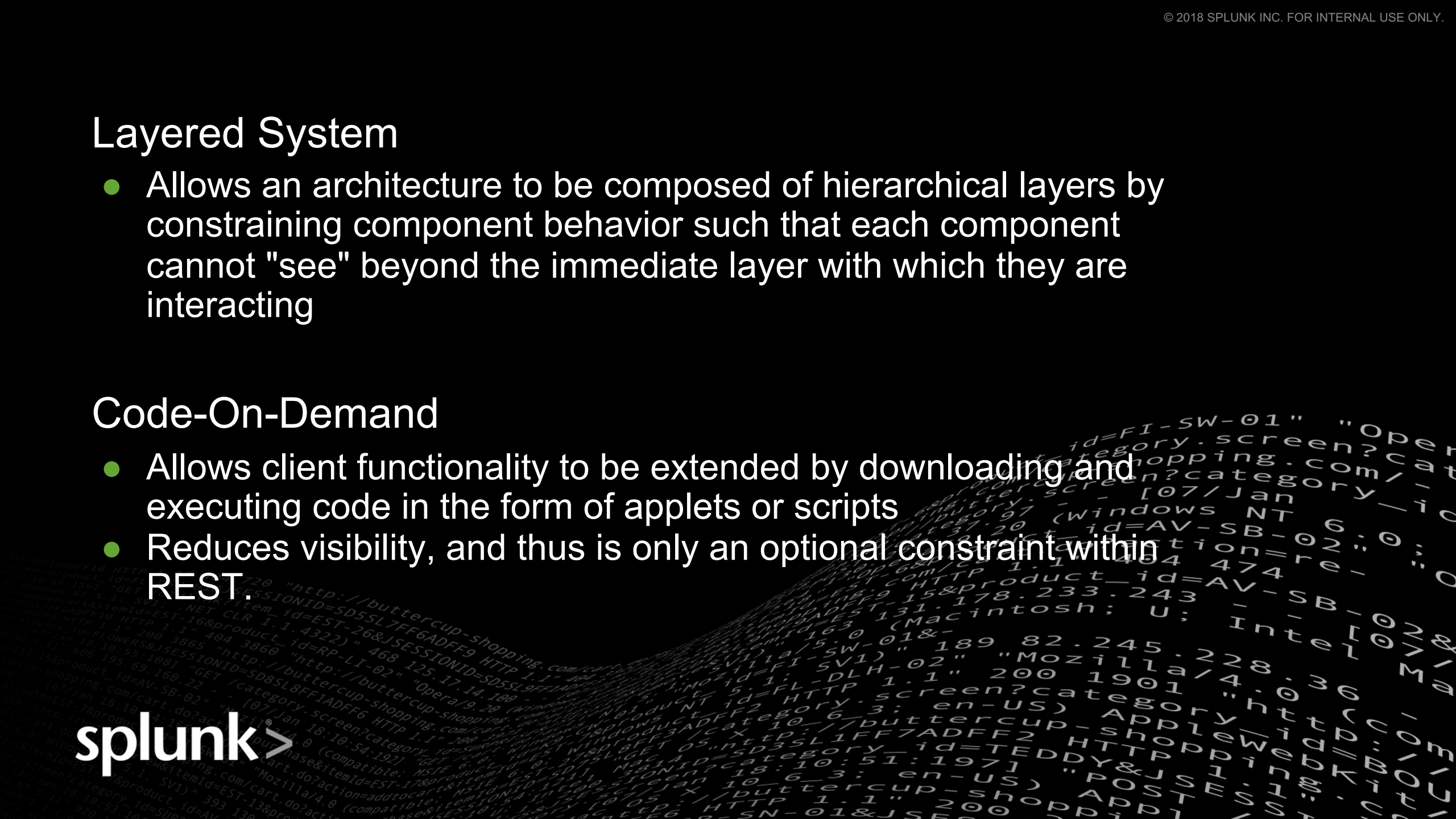


Layered System

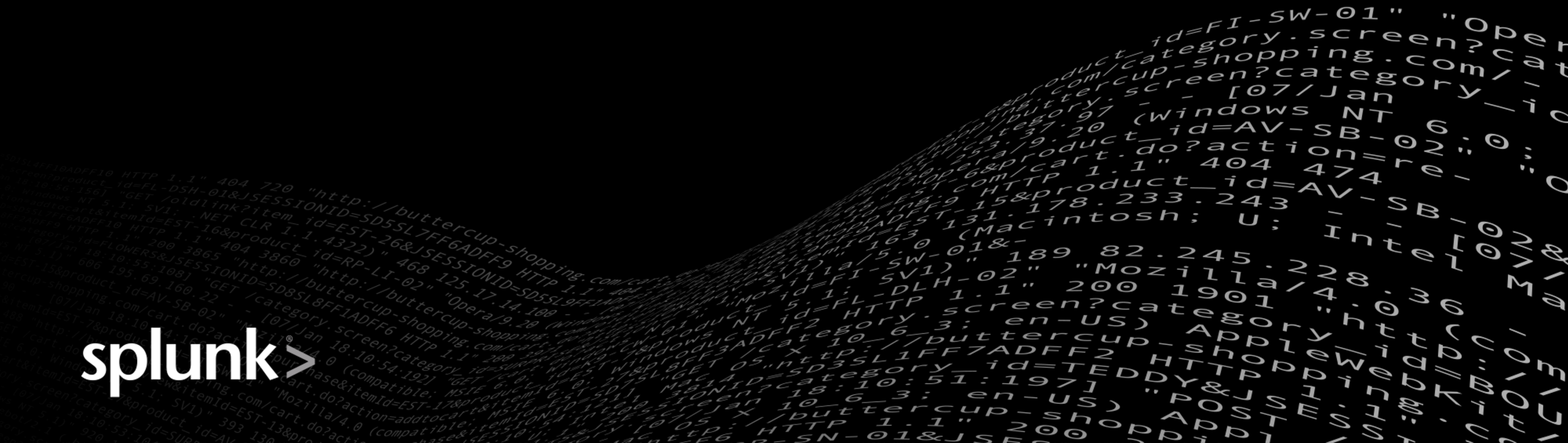
- Allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting

Code-On-Demand

- Allows client functionality to be extended by downloading and executing code in the form of applets or scripts
- Reduces visibility, and thus is only an optional constraint within REST.



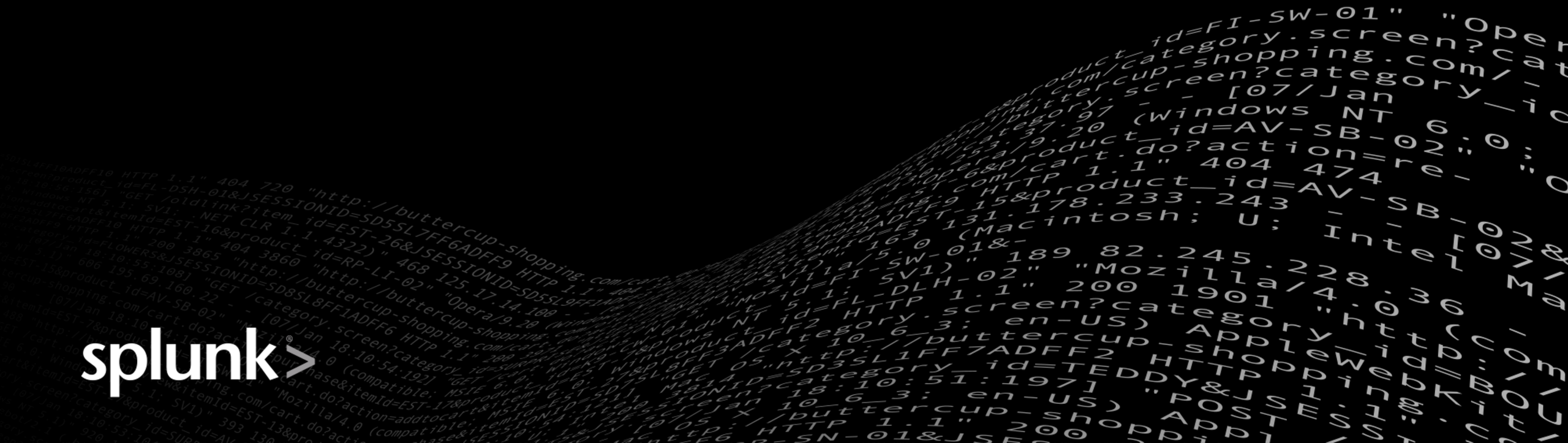
Good REST API Design



Start with your API

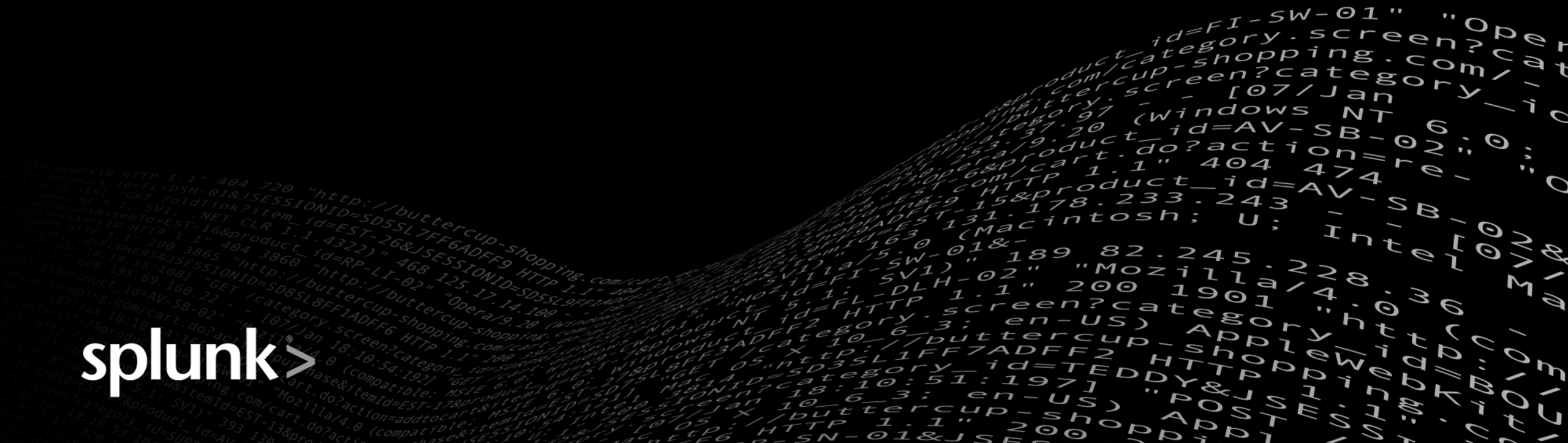
- API design should not dictate your architecture, but it may likely inform it
- If you are working with an existing system, establish your API as a “facade”
- Think about the capabilities that need to be exposed to your client

“It’s all the same. Only the names will change.”
-Jon Bon Jovi



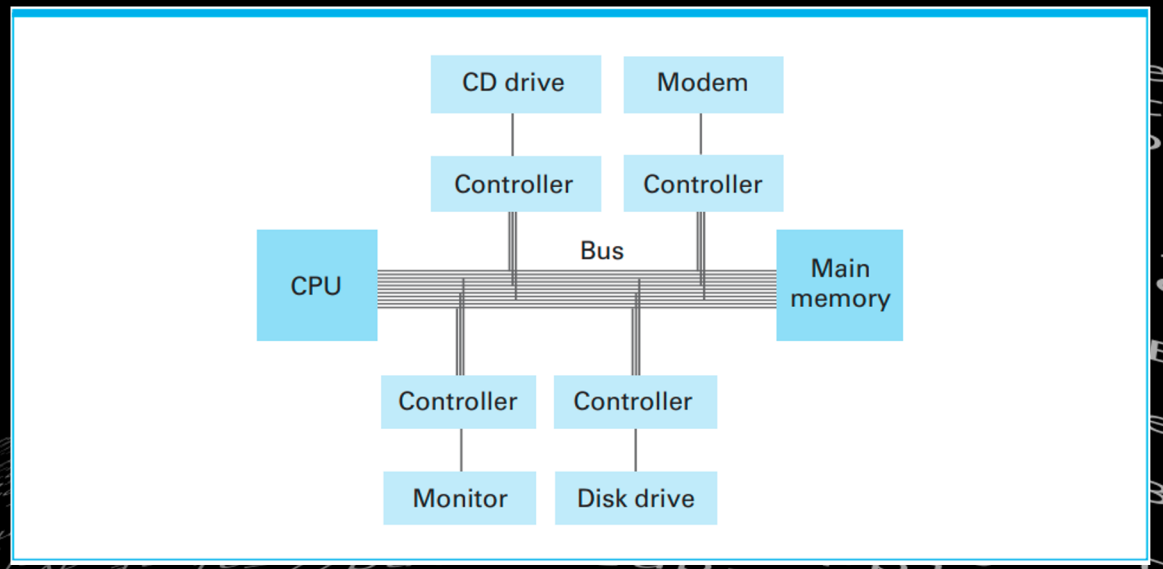
Computational System Design Models

- Microcomputer architecture
- Unix operating system



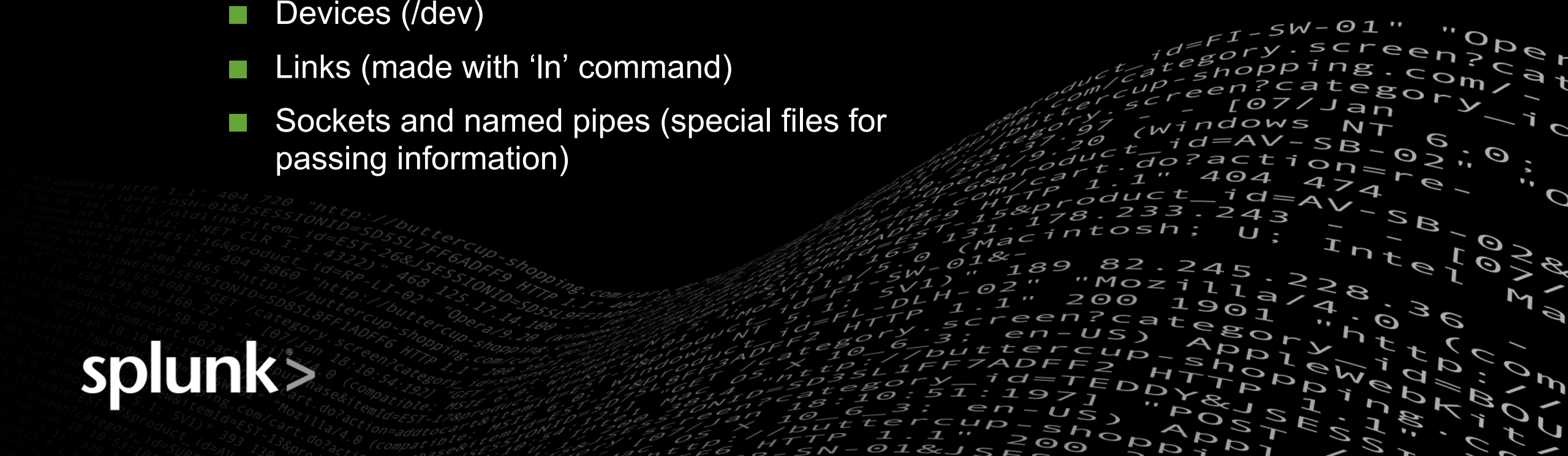
Computational System Design Models

- Microcomputer architecture
  - The core processing unit (CPU/ALU) accesses all resources through an address bus.
    - Memory
    - Controllers



Computational System Design Models

- Unix operating system
  - Everything is a file
    - Directories (/home/user)
    - Devices (/dev)
    - Links (made with ‘ln’ command)
    - Sockets and named pipes (special files for passing information)



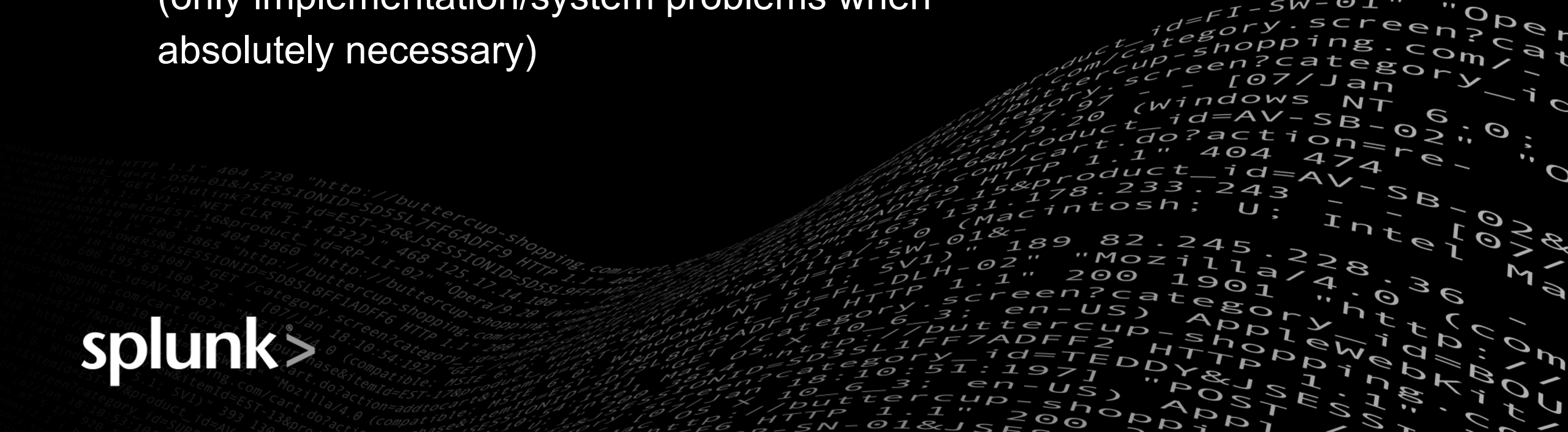
REST Design Tips

- Model your process not the implementation
- Design resources and representations that reflect your business logic
- Design resources as nouns (not verbs or prepositions)
- Design resources and representations to be composable

splunk>

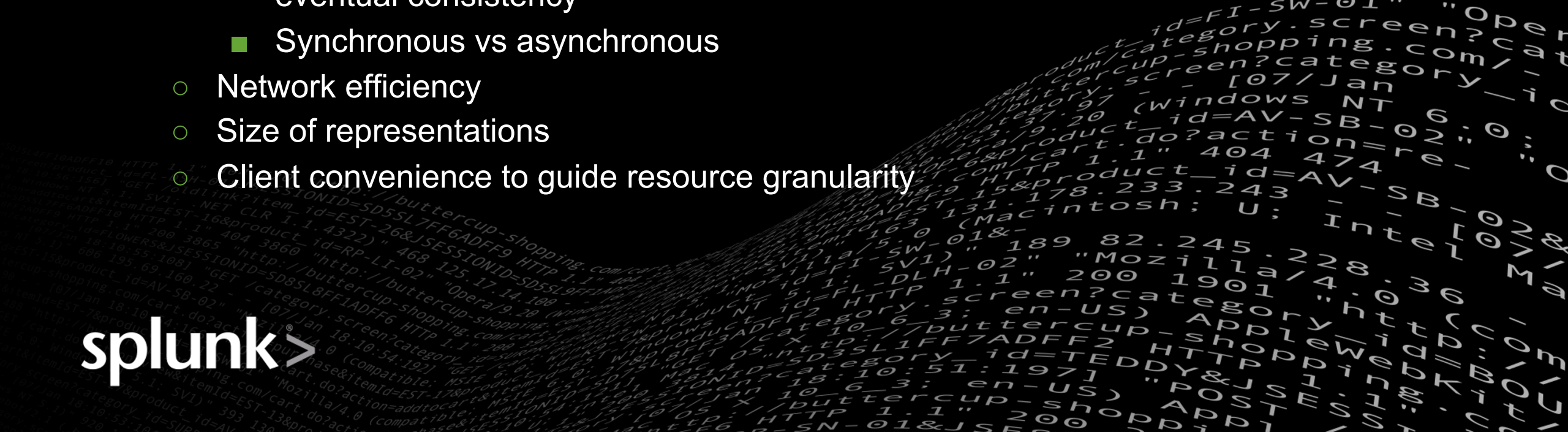
Model your process not the implementation

- Critical processes (i.e. resource ID assignment) should be isolated to the server
- Errors should reflect business logic problems (only implementation/system problems when absolutely necessary)



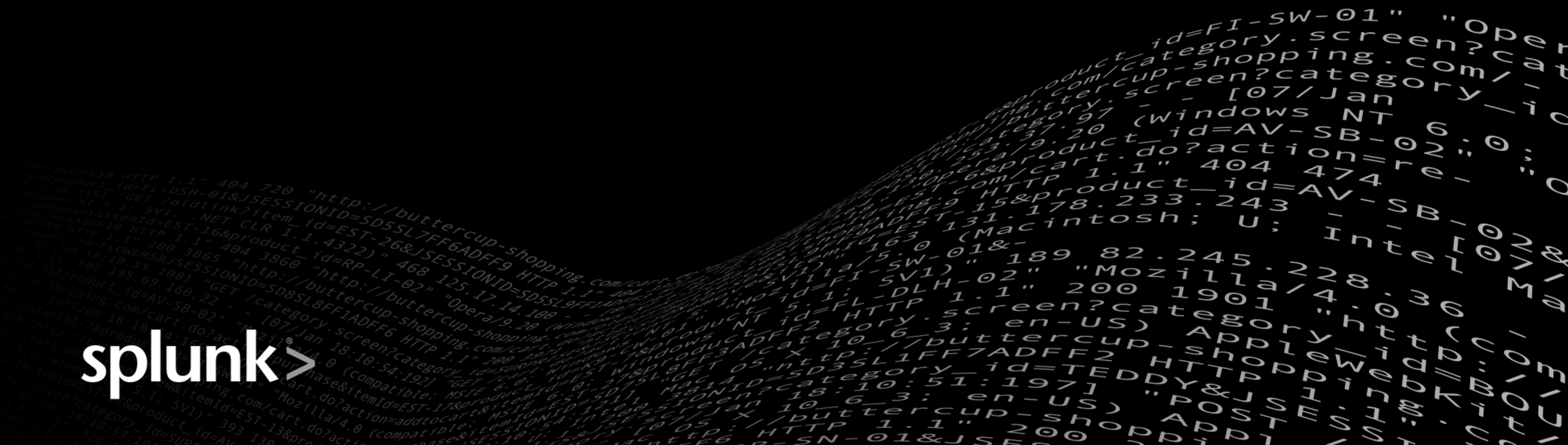
Design resources and representations that reflect your business logic

- Keep in mind
  - Creation and change properties
    - Atomic transaction/immediate consistency vs eventual consistency
    - Synchronous vs asynchronous
  - Network efficiency
  - Size of representations
  - Client convenience to guide resource granularity



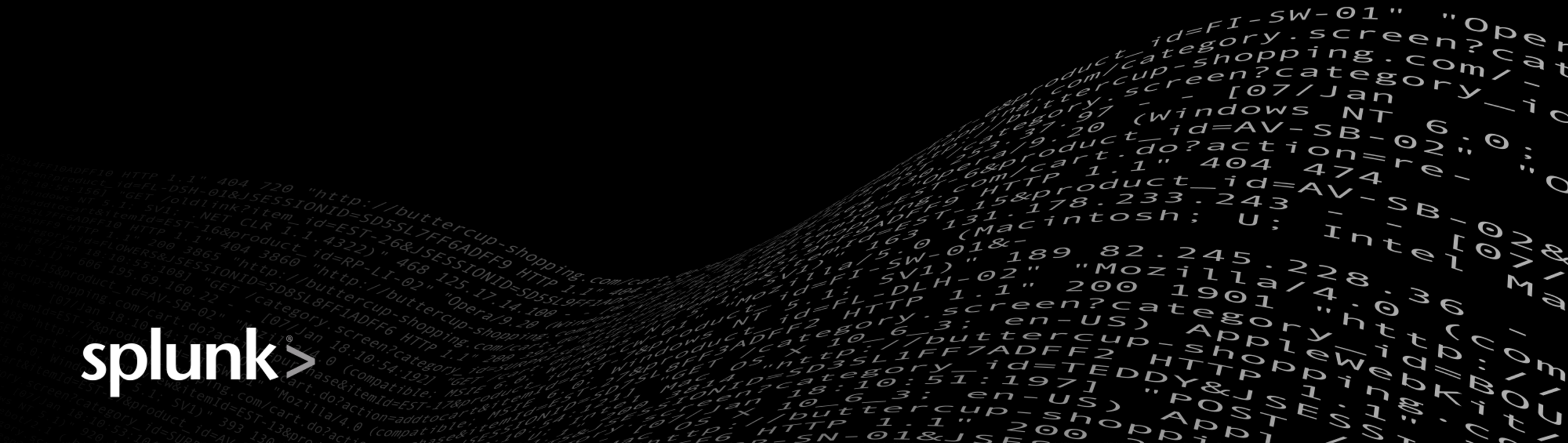
Design resources and representations that reflect your business logic

- Avoid modeling
  - Your persistence schema
  - Other traits of your implementation



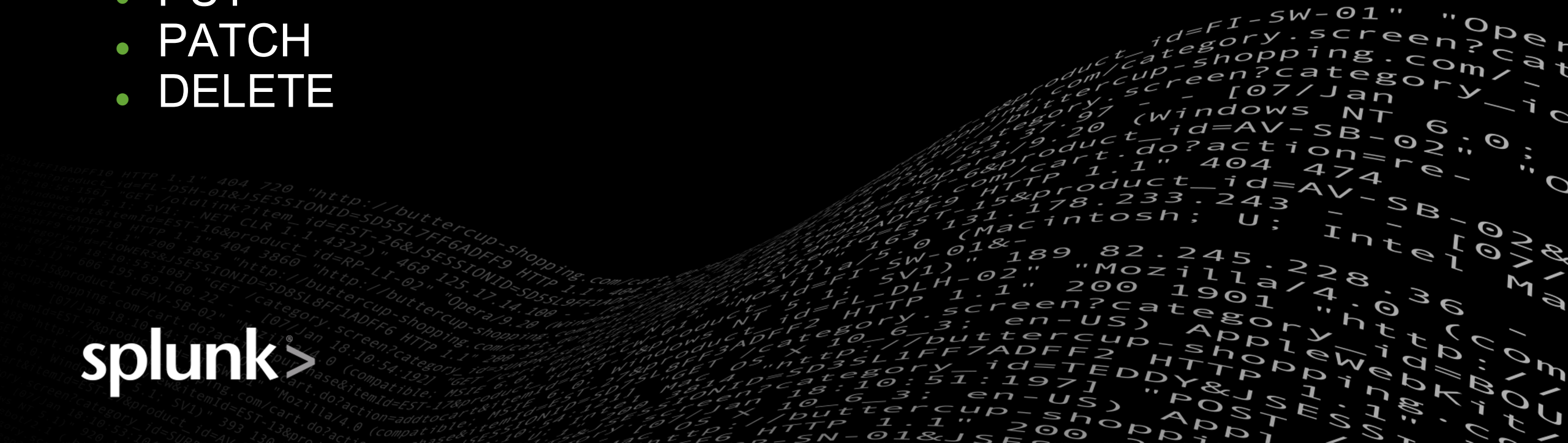
Design resources as nouns (not verbs or prepositions)

- Resources can be designed in ways similar to objects in OOP (not necessarily nouns in the strict sense)



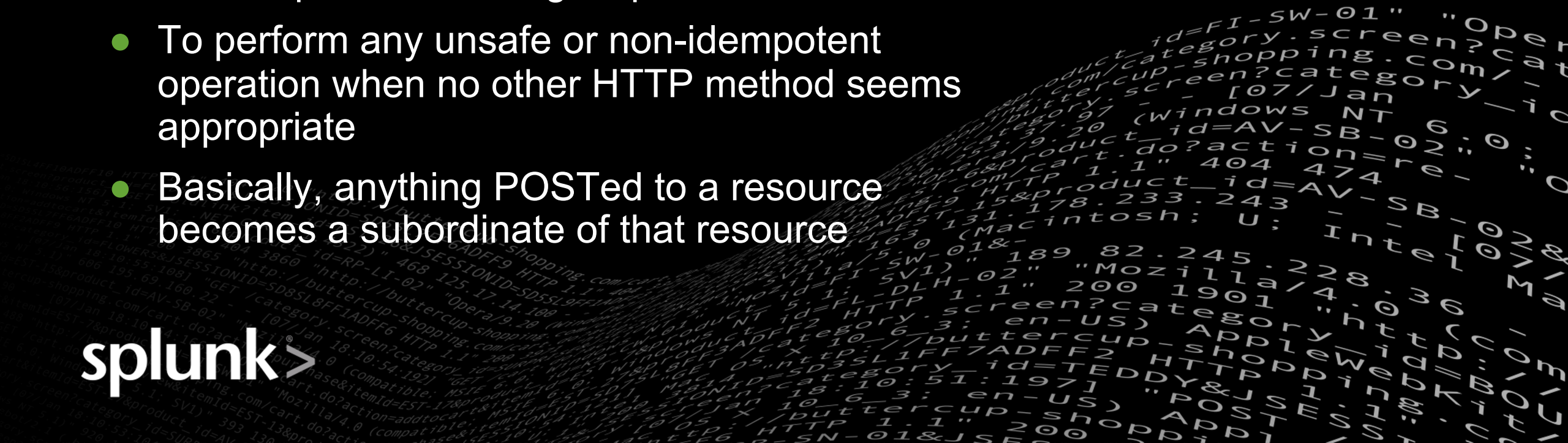
Know your verbs, their design, and their capabilities (the Uniform Interface)

- POST
- GET
- PUT
- PATCH
- DELETE



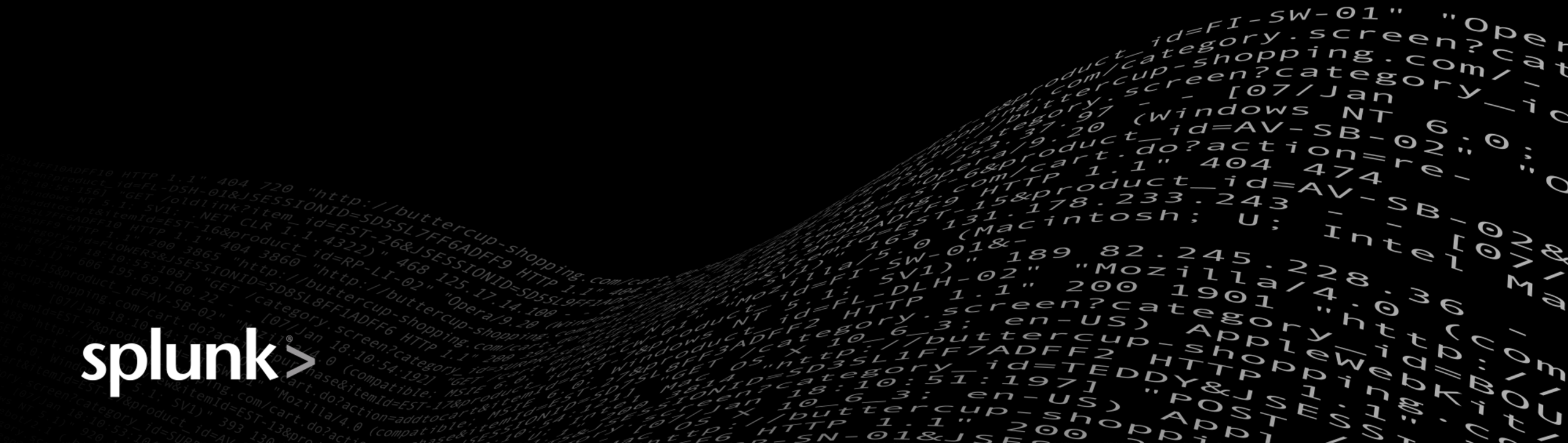
Use POST

- To create a new resource, using the resource as a factory
- To submit input to any process that would be asynchronous
- To run queries with large inputs
- To perform any unsafe or non-idempotent operation when no other HTTP method seems appropriate
- Basically, anything POSTed to a resource becomes a subordinate of that resource



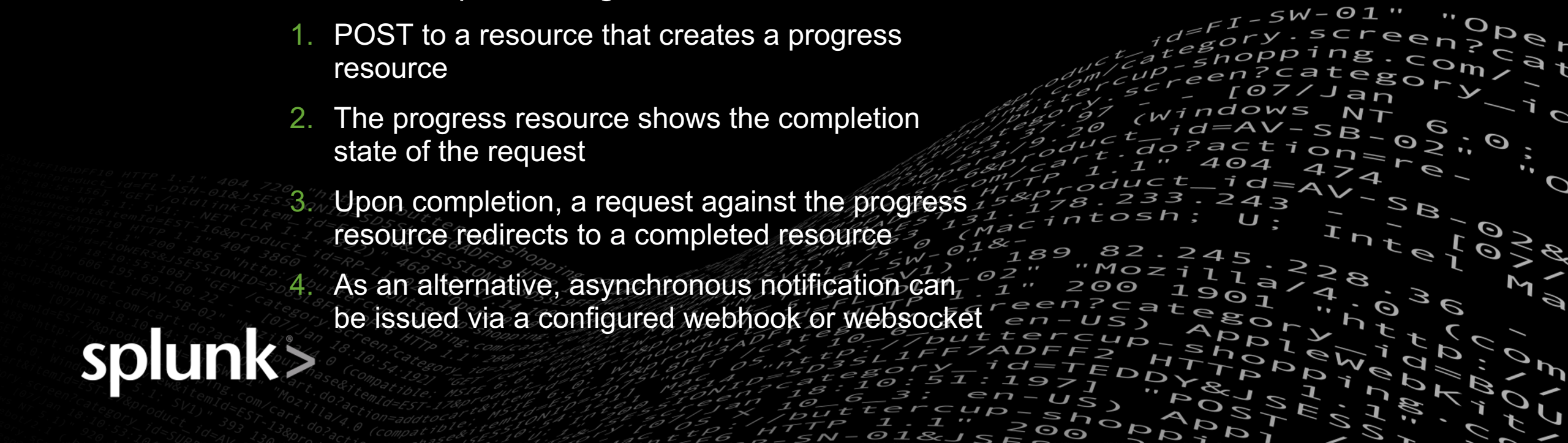
Use POST

- To create a new resource, using the resource as a factory
  - System specific values (i.e. unique IDs) should be generated by the resource server to prevent business logic from implemented in the client



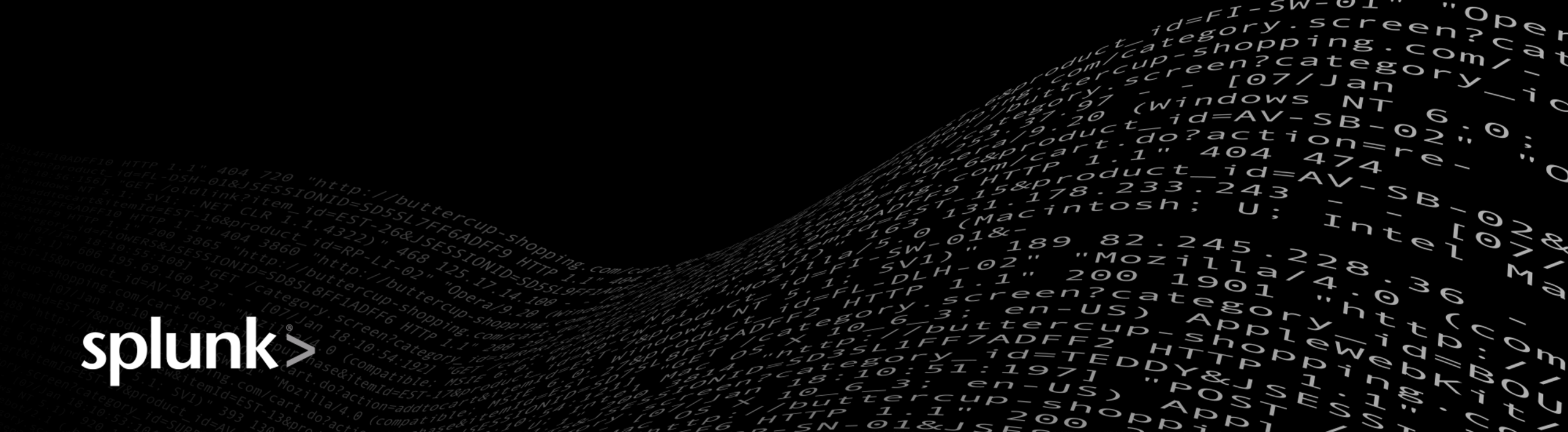
Use POST

- To submit input to any process that would be asynchronous
  - Commands
  - CRUD for resources that are eventually consistent
    - Use a batch processing model
      1. POST to a resource that creates a progress resource
      2. The progress resource shows the completion state of the request
      3. Upon completion, a request against the progress resource redirects to a completed resource
      4. As an alternative, asynchronous notification can be issued via a configured webhook or websocket



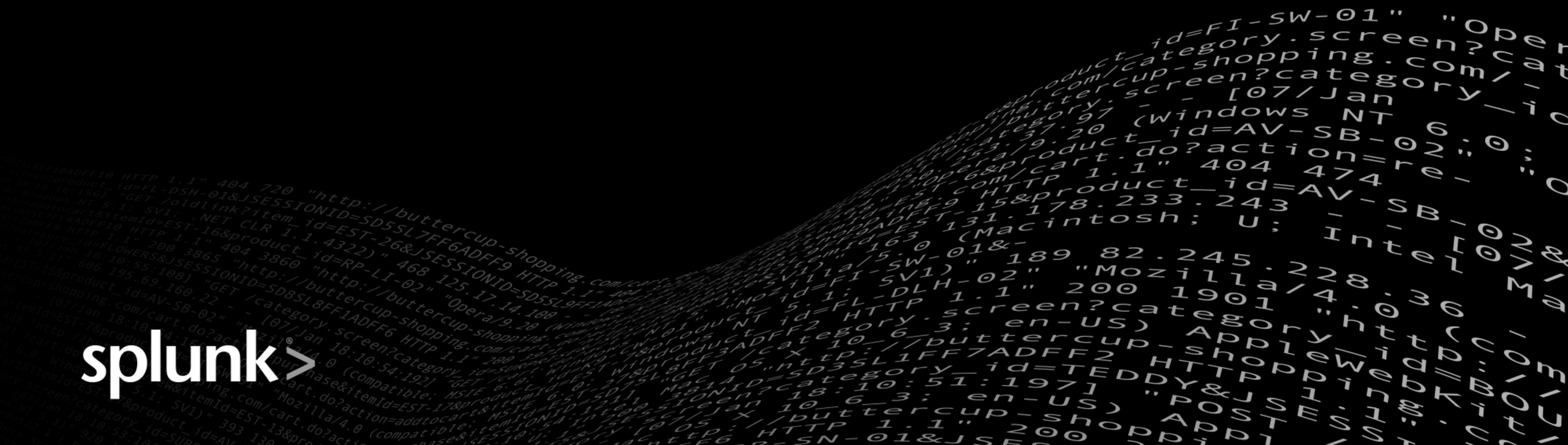
Use POST

- To run queries with large inputs
  - Safer than using query variables in a URI
    - Used to be used as a server attack method, inducing a buffer overflow



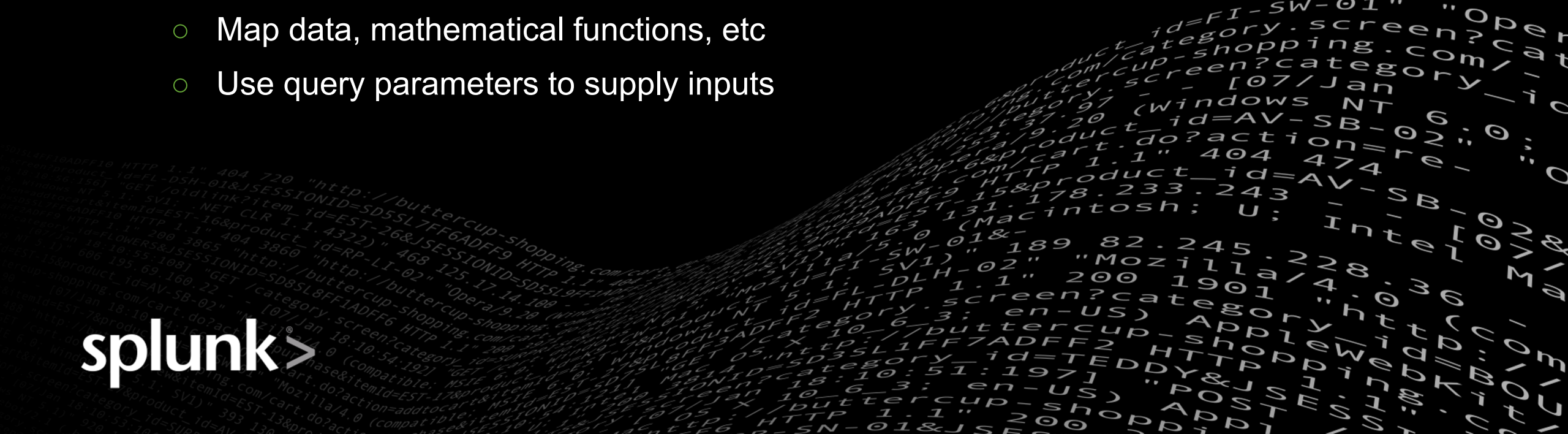
Use POST

- To perform any unsafe or non-idempotent operation when no other HTTP method seems appropriate
  - Processes that yield side effects are executed this way



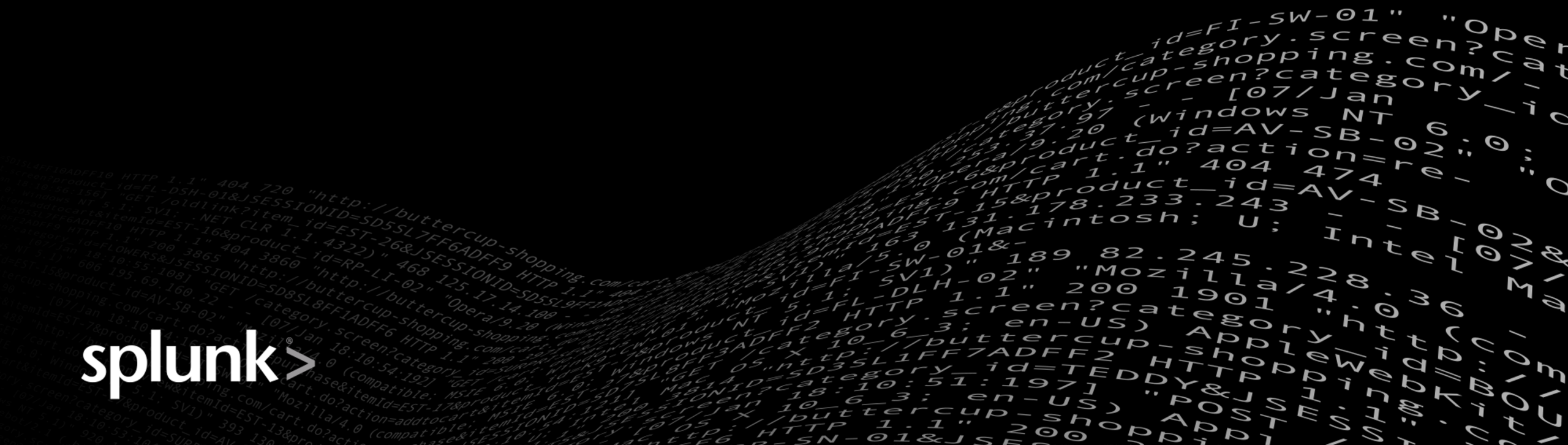
Using GET

- For safe and idempotent (no side effects) information retrieval of resources
- To fetch a representation containing the output of the processing function or data over a continuous domain
  - Map data, mathematical functions, etc
  - Use query parameters to supply inputs



Using PUT

- To update named individual resources in an idempotent (no side effects) way
  - Best for resources that already exist on the server
  - Process should be atomic (immediately consistent)

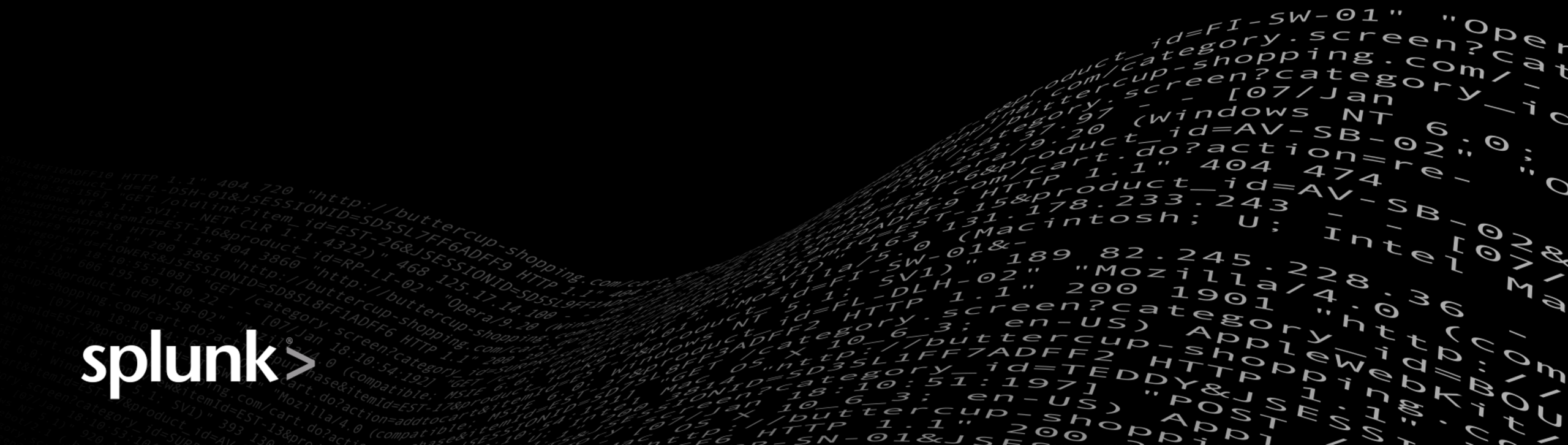


Using PATCH

- To update resources in an idempotent, non-constrained way
- Do NOT use
  - If the underlying operation is not atomic/non transactional
  - If the input is constrained to a well-defined, unchanging representation
    - Use PUT instead

Using DELETE

- To remove individual resources from visibility
  - Can be removed from the system
  - Can be archived
- May be implemented asynchronously



Let's Build an API

We're building a system that keeps track of users and their mailing addresses. The user information consists of…

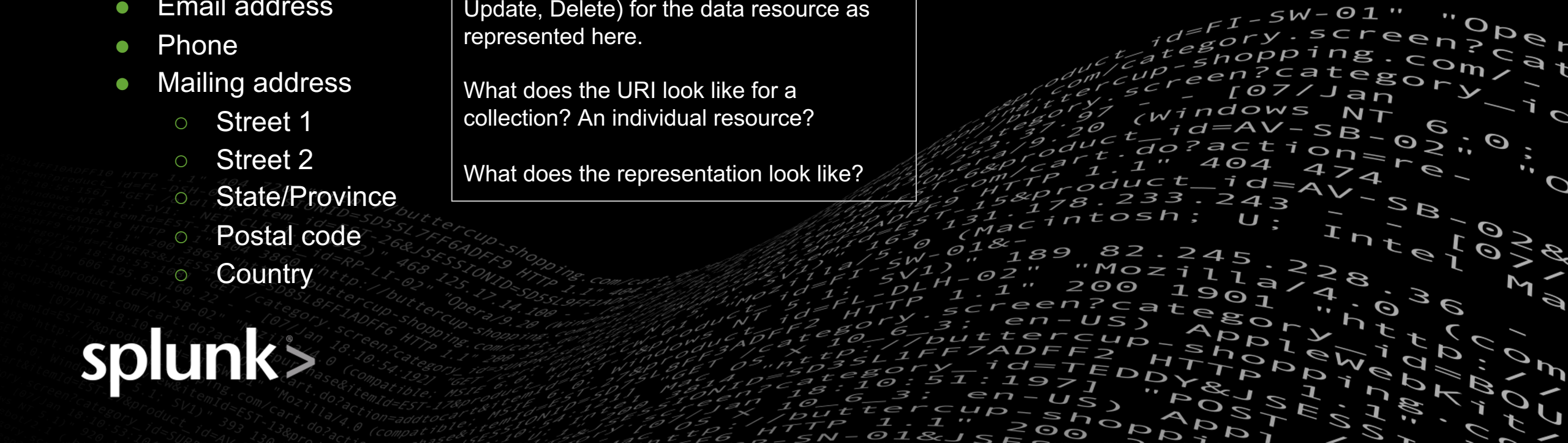
- First name
- Last name
- Email address
- Phone
- Mailing address
  - Street 1
  - Street 2
  - State/Province
  - Postal code
  - Country

Part 1:

Implement CRUD (Create, Retrieve, Update, Delete) for the data resource as represented here.

What does the URI look like for a collection? An individual resource?

What does the representation look like?



Let's Build an API

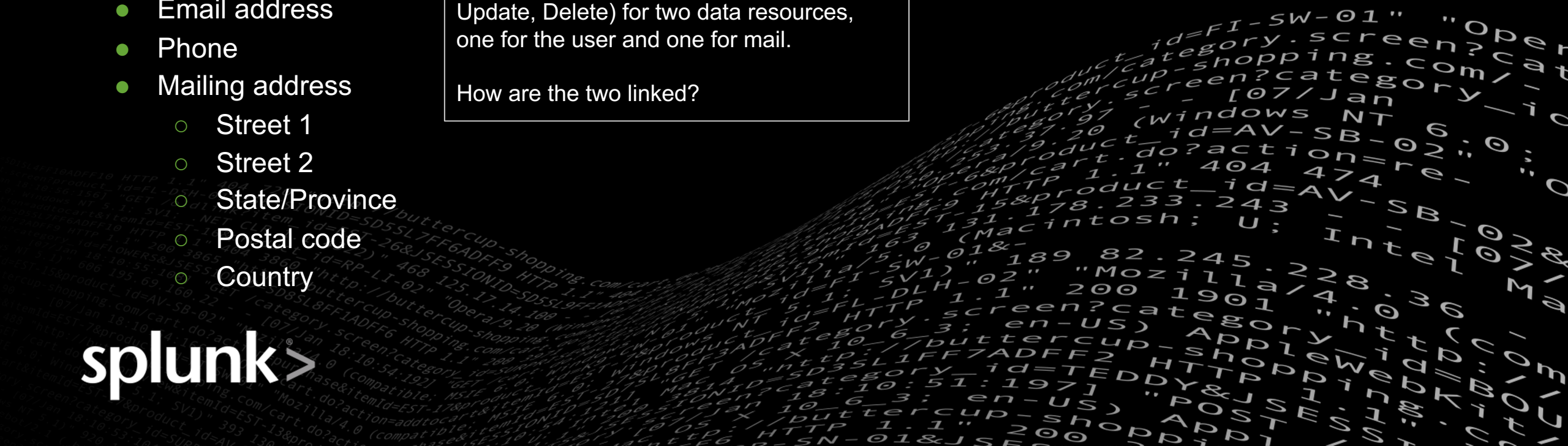
We're building a system that keeps track of users and their mailing addresses. The user information consists of…

- First name
- Last name
- Email address
- Phone
- Mailing address
  - Street 1
  - Street 2
  - State/Province
  - Postal code
  - Country

Part 2:

Implement CRUD (Create, Retrieve, Update, Delete) for two data resources, one for the user and one for mail.

How are the two linked?



Design resources and representations to be composable

- Resource representations can be composed into a larger resource for easier access
  - This should be driven by monitoring client usage patterns
- Remember the user/mail example