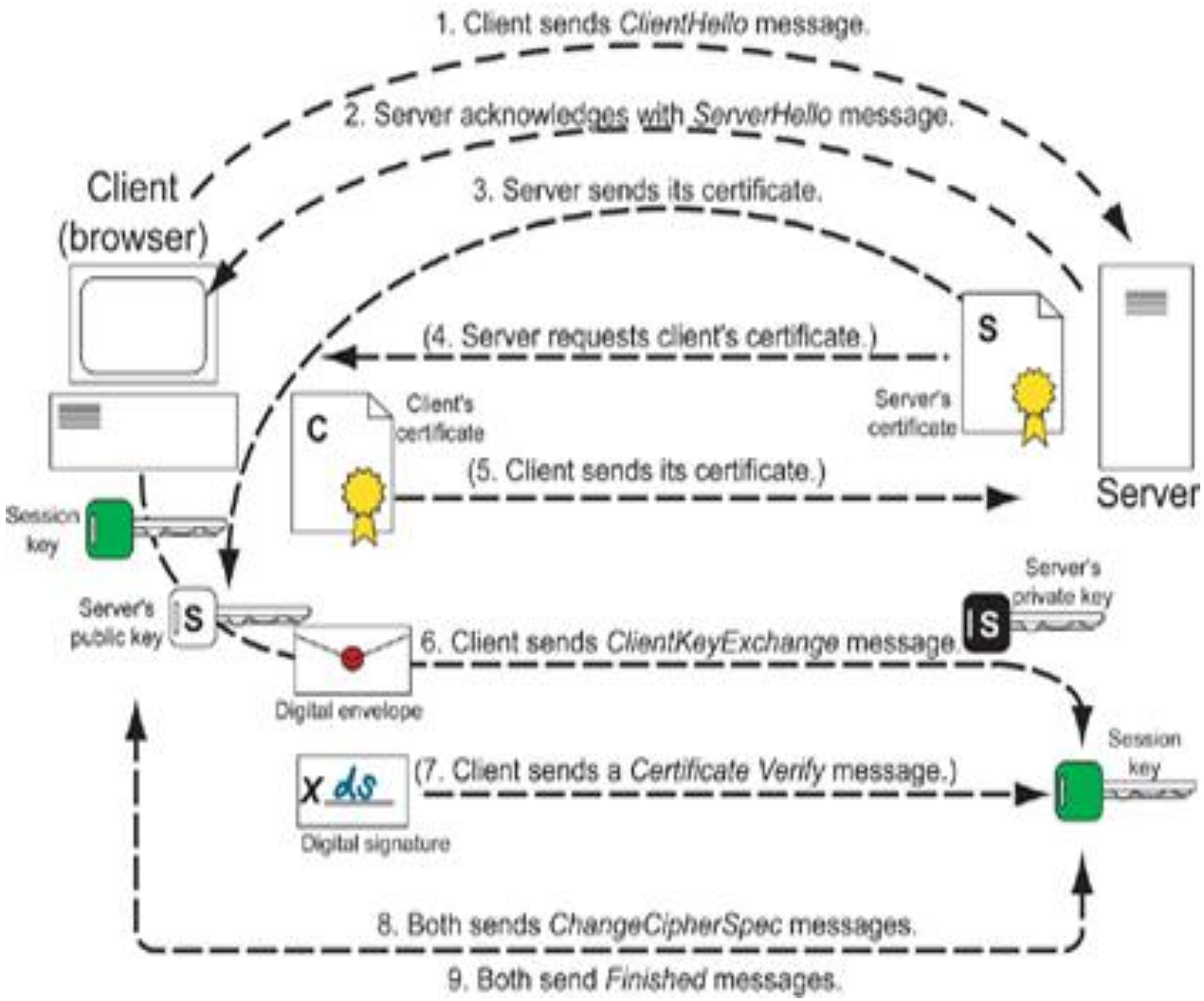
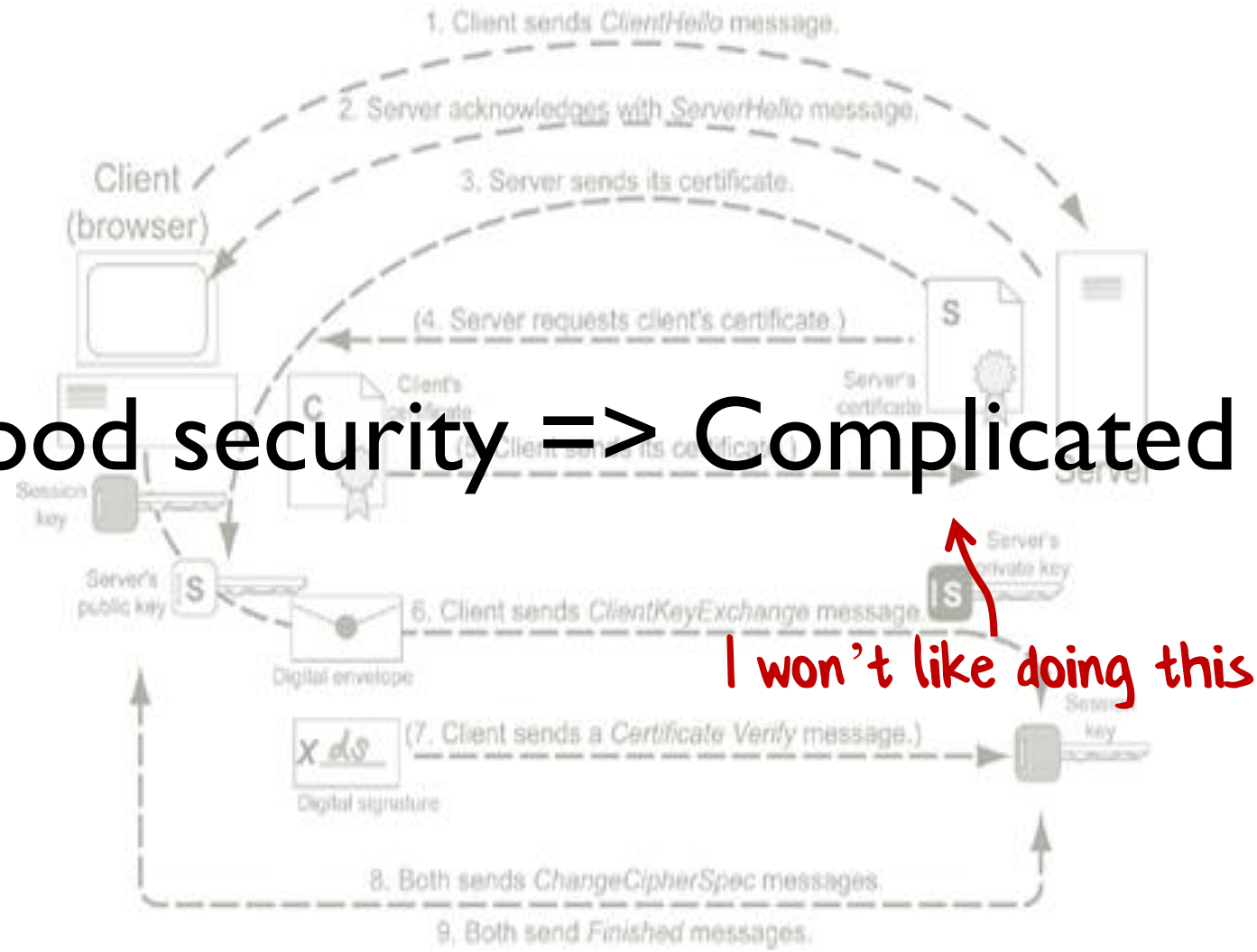
Designing with capabilities for fun and profit

@ScottWlaschin

fsharpforfunandprofit.com/cap

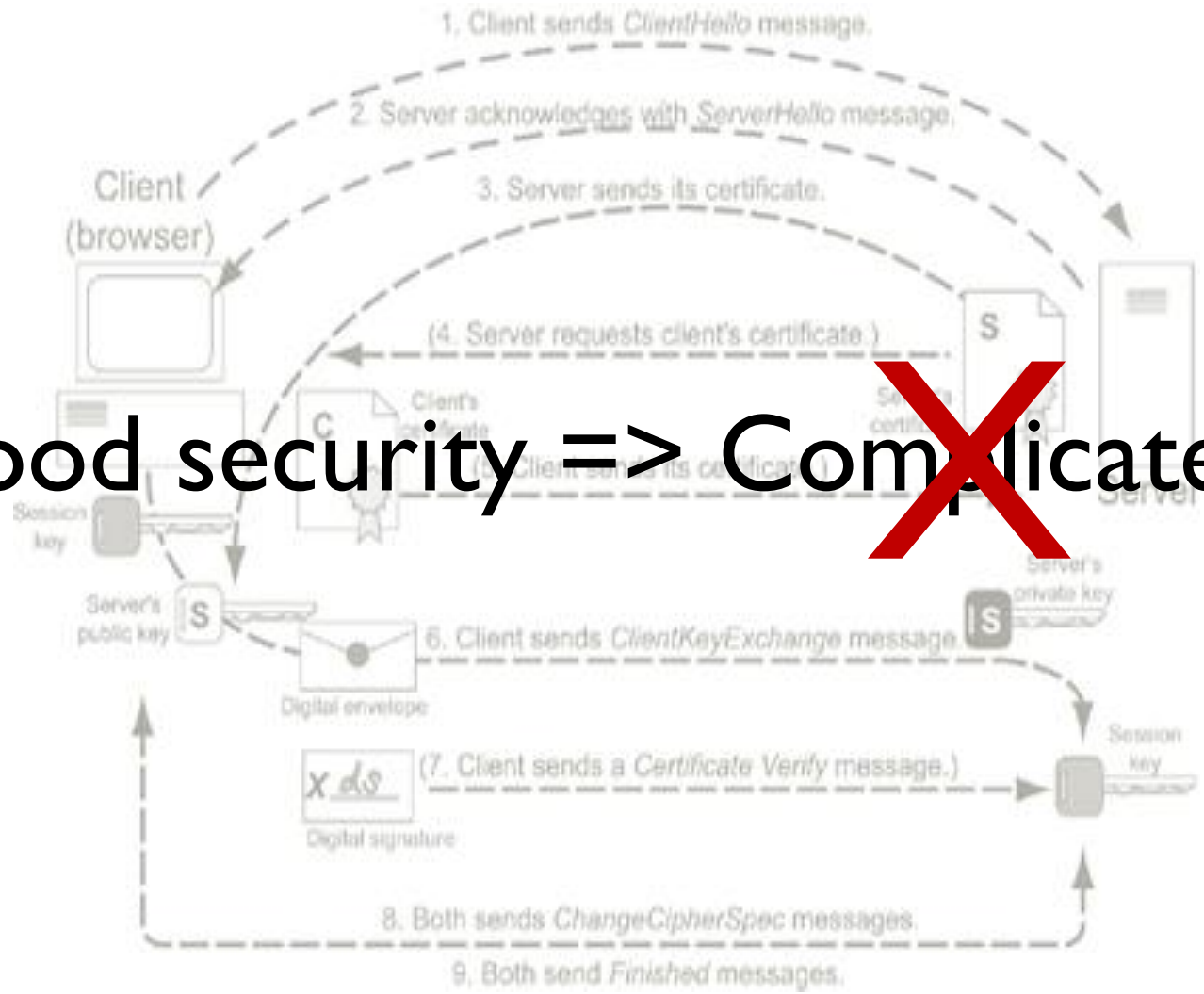

1. Client sends ClientHello message.
2. Server acknowledges with ServerHello message.
3. Server sends its certificate.
Client
(browser)
(4. Server requests client's certificate.)
S
Server's
certificate
C
Client's
certificate
(5. Client sends its certificate.)
Server
Session
key
Server's
public key
S
Server's
private key
S
6. Client sends ClientKeyExchange message.
Digital envelope
Session
key
(7. Client sends a Certificate Verify message.)
Digital signature
8. Both sends ChangeCipherSpec messages.
9. Both send Finished messages.

Good security => Complicated



I won't like doing this

Good security => Complicated



Good security == Good design

Security for free!
...which is good,
because I'm lazy

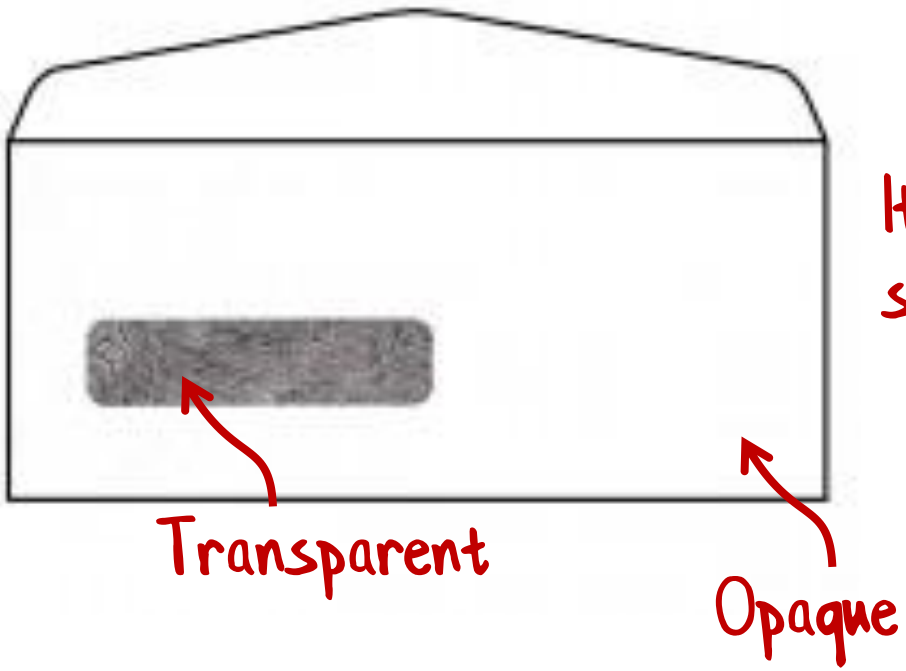
I like doing this

Good security == Good design

The topic of this talk

Not about OAuth, JWT etc

Disclaimer – I am not a security expert



It's all about security, right?

Please deliver this letter

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, Temporibus autem quibus Dacei Megasystems Tech Inc necessitatibust aut officiis debitis auteo 2799 E Dragam Suite 7 quisquam saepe Itaque enieti Los Angeles CA 90002 ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?

A counterexample

Please deliver this letter

Sed ut perspiciatis unde omnis iste natus error sit voluptatem
accusantium doloremque laudantium, totam rem aperiam,
eaque ipsa quae ab illo inventore veritatis et quasi architecto
beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem
quia voluptas sit aspernatur aut odit aut fugit, sed quia
consequuntur magni dolores eos qui ratione voluptatem sequi
nesciunt. Neque porro quisquam est, qui dolorem ipsum quia
dolor sit amet, consectetur, adipisci velit, sed quia non
numquam eius modi tempora incidunt ut labore et dolore
magnam aliquam quaerat voluptatem. Ut enim ad minima
veniam, quis nostrum exercitationem ullam corporis suscipit
laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis
autem vel eum iure reprehenderit qui in ea voluptate velit esse
quam nihil molestiae consequatur, Temporibus autem quibus
Dacei necessitatibust aut officiis debitis
auteo quisquam saepe Itaque
enieti ut et voluptates repudiandae sint
et molestiae non recusandae. Itaque earum rerum nic tenetur a
sapiente delectus, ut aut reiciendis voluptatibus maiores alias
consequatur aut perferendis doloribus asperiores repellat.
Neque porro quisquam est, qui dolorem ipsum quia dolor sit
amet, consectetur, adipisci velit, sed quia non numquam eius
modi tempora incidunt ut labore et dolore magnam aliquam
quaerat voluptatem. Ut enim ad minima veniam, quis nostrum
exercitationem ullam corporis suscipit laboriosam, nisi ut
aliquid ex ea commodi consequatur?

Megasystems Tech Inc
2799 E Dragam Suite 7
Los Angeles CA 90002

It's not just about security...

...hiding irrelevant information is good design!

EVOLUTION OF AN API

Evolution of an API
(from the security point of view)

Say that the UI needs to set a configuration option (e.g. DontShowThisMessageAgain)

How can we stop a malicious caller doing bad things?

Version 1
Give the caller the configuration file name

API

```
interface IConfiguration
{
    string GetConfigFilename();
}
```

Caller

```
var filename = config.GetConfigFilename();
// open file
// write new config
// close file
```

☹ A malicious caller has the ability to open and write to any file on the filesystem

Version 2
Give the caller a TextWriter

API

```
interface IConfiguration
{
    TextWriter GetConfigWriter();
}
```

We control which file is opened

Caller

```
var writer = config.GetConfigWriter();
// write new config
```

☹ A malicious caller can corrupt the config file

Version 3
Give the caller a key/value interface

API

```
interface IConfiguration
{
  void SetConfig(string key, string value);
}
```

Caller

```
config.SetConfig(
  "DontShowThisMessageAgain", "True");
```

☹ A malicious caller can set the value to a non-boolean

Version 4
Give the caller a domain-centric interface

API

```
enum MessageFlag {
   ShowThisMessageAgain,
   DontShowThisMessageAgain
   }

interface IConfiguration
{
    void SetMessageFlag(MessageFlag value);
    void SetConnectionString(ConnectionString value);
    void SetBackgroundColor(Color value);
}
```

☹ What's to stop a malicious caller changing the connection string when they were only supposed to set the flag?

Version 5
Give the caller only the interface they need

API

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

☺ The caller can *only* do the thing we allow them to do.

Security spectrum

Principle of Least Authority (POLA)

Can't get your work done

Just right

Potential for abuse

Too little information passed in

Too much information passed in

Evolution of the same API
(from the design point of view)

Design Version 1
Give the caller the configuration file name

API

```
interface IConfiguration
{
    string GetConfigFilename();
}
```

Bad design: Using a filename means we limit ourselves to file-based config files.

Better design: A TextWriter would make the design more mockable.

Design Version 2
Give the caller a TextWriter

API

```
interface IConfiguration
{
    TextWriter GetConfigWriter();
}
```

Bad design: Using a TextWriter means exposing a specific storage format

Better design: A generic KeyValue store would make implementation choices more flexible.

Design Version 3
Give the caller a key/value interface

API

```
interface IConfiguration
{
  void SetConfig(string key, string value);
}
```

Bad design: A KeyValue store using strings means possible bugs. So we need to write validation and tests for that ☹

Better design: A statically typed interface means no corruption checking code. ☺

Design Version 4
Give the caller a domain-centric interface

API

```
enum MessageFlag {
   ShowThisMessageAgain,
   DontShowThisMessageAgain
   }

interface IConfiguration
{
    void SetMessageFlag(MessageFlag value);
    void SetConnectionString(ConnectionString value);
    void SetBackgroundColor(Color value);
}
```

Bad design: An interface with too many methods violates the ISP.

Better design: Reduce the number of available methods to one!

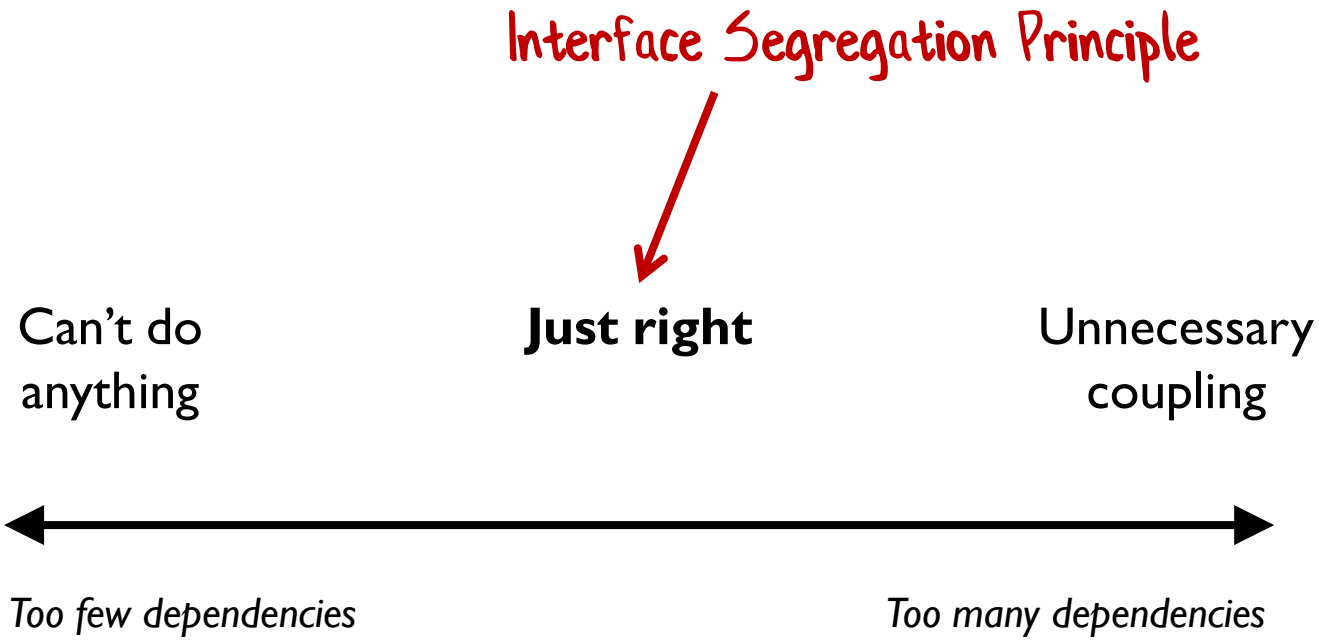
Design Version 5
Give the caller only the interface they need

API

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

Good design: The caller has no dependencies on anything else. Bonus: easy to mock! ☺

Design spectrum



OO Design Guidelines

Interface Segregation Principle
Single Responsibility Principle, etc

Ak.a. Minimize your surface area
(to reduce coupling, dependencies, etc)

Security Guidelines

Principle of Least Authority (POLA)

Ak.a. Minimize your surface area
(to reduce chance of abuse)

Good security => Good design

Good design => Good security

INTRODUCING “CAPABILITIES”

Capability based design

- In a cap-based design, the caller can only do exactly one thing -- a "capability".
- In the example, the caller has a capability to set the message flag, and that's all.

Prevents both maliciousness and stupidity!

A one method interface is a capability

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

A capability

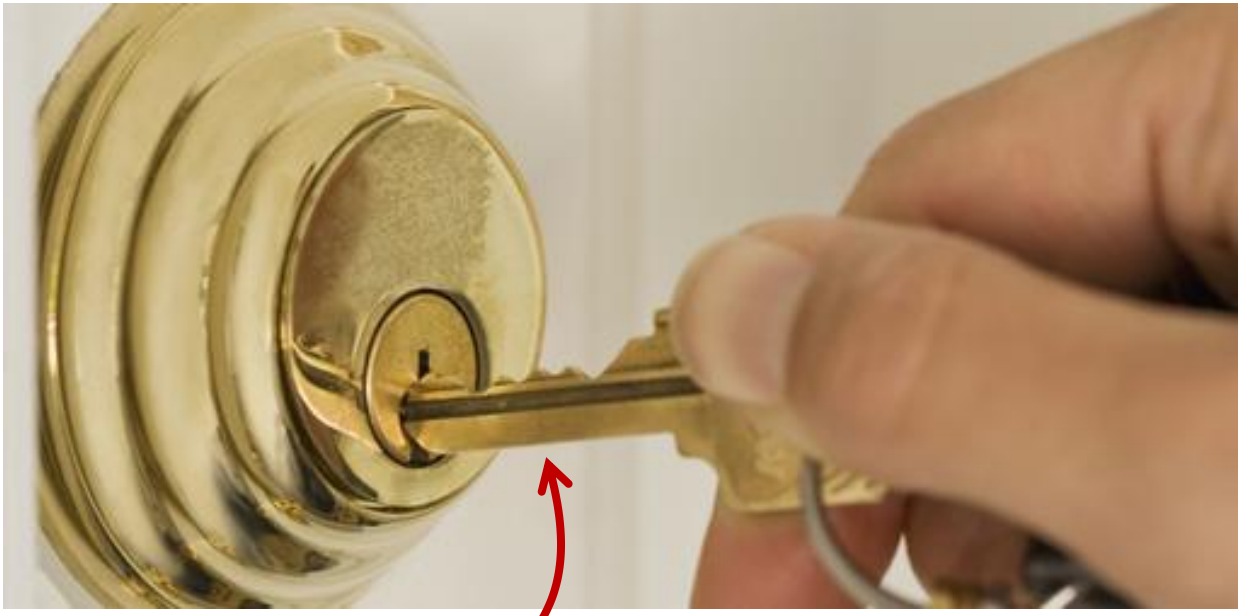
A one method interface is a function

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

```
Action<MessageFlag> messageFlagCapability
```

Pro tip: Functions are a great way of implementing capabilities

Capability-based security in the real world



A capability

Access is based on what you have

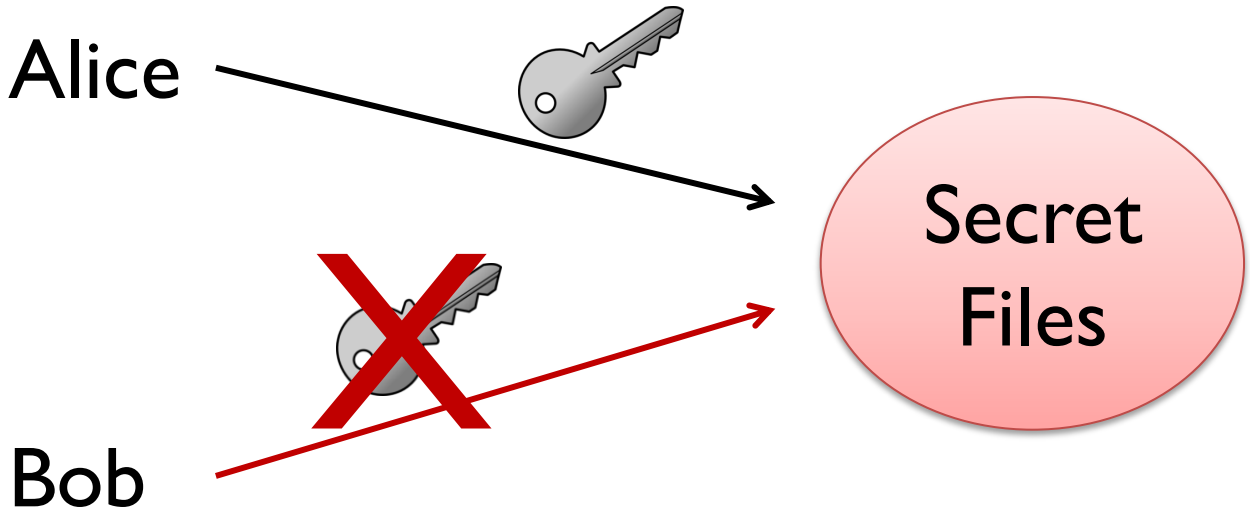
A set of capabilities

What does "access-control" mean?

- **Preventing** any access at all.
- **Limiting** access to some things only.
- **Revoking** access when you are no longer allowed.
- **Granting** and delegating access to some subset of things.

It's not always about saying no!

Using capabilities to control access to a resource

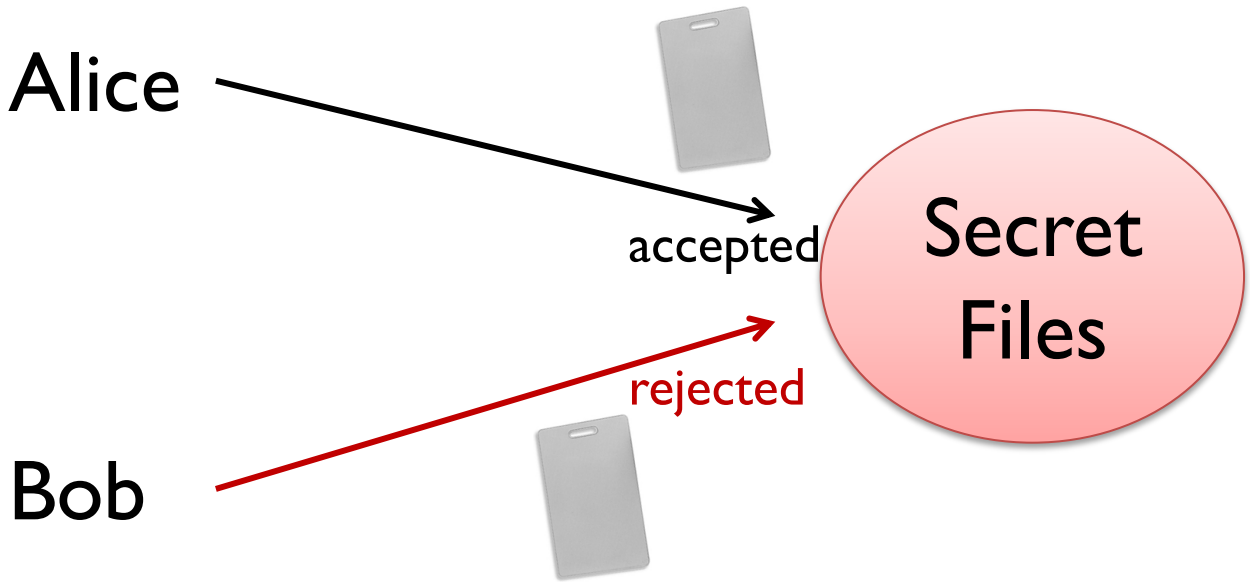


Alternative for access control

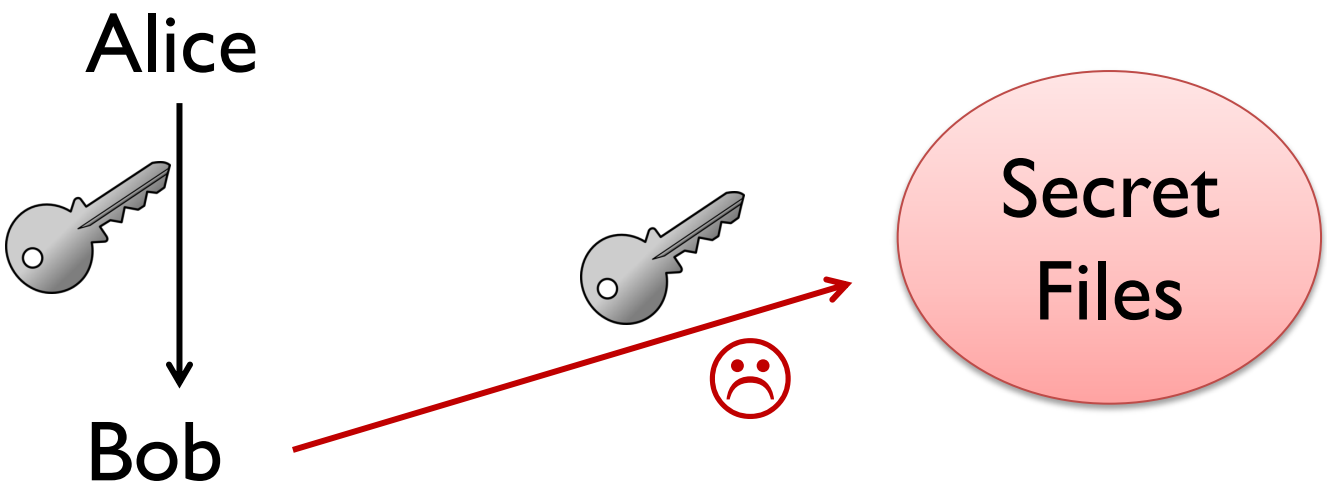
Authentication + RBAC

==> Access based on who you are

Using auth/RBAC to control access to a resource

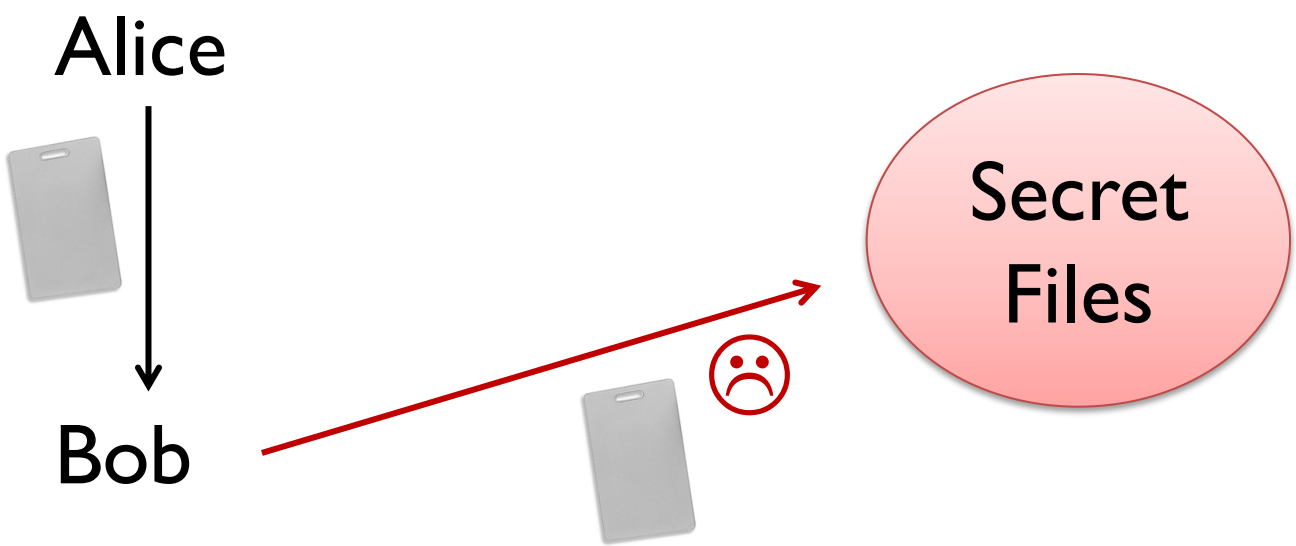


A weakness in capabilities



Anyone with the key can get access.

A weakness in auth/RBAC



A weakness in *any* security system!

Do you trust Alice?

Alice

Secret Files

Then you also trust anyone whom Alice trusts too...

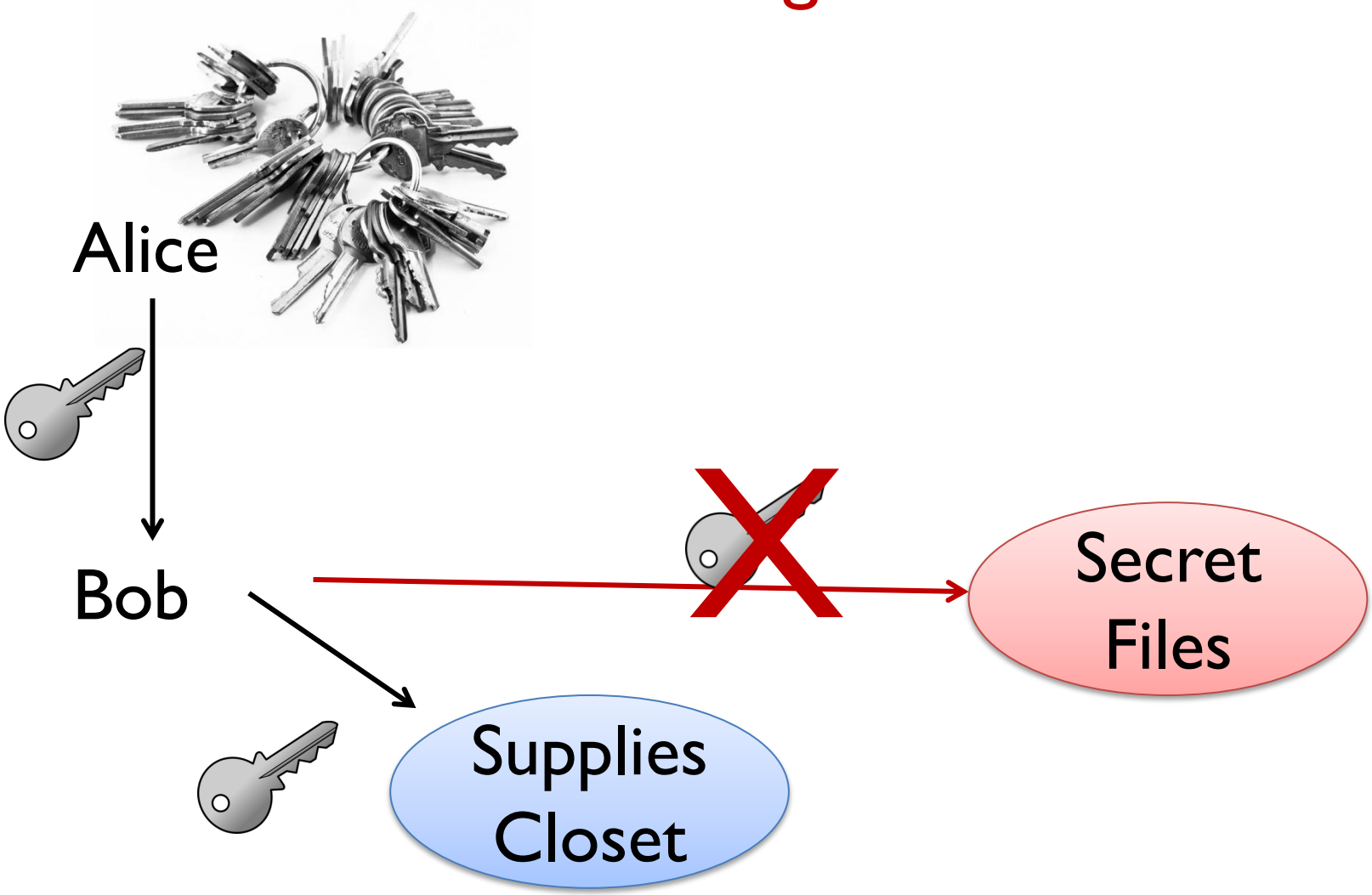
Bob

(Or whom Alice can be tricked into trusting)

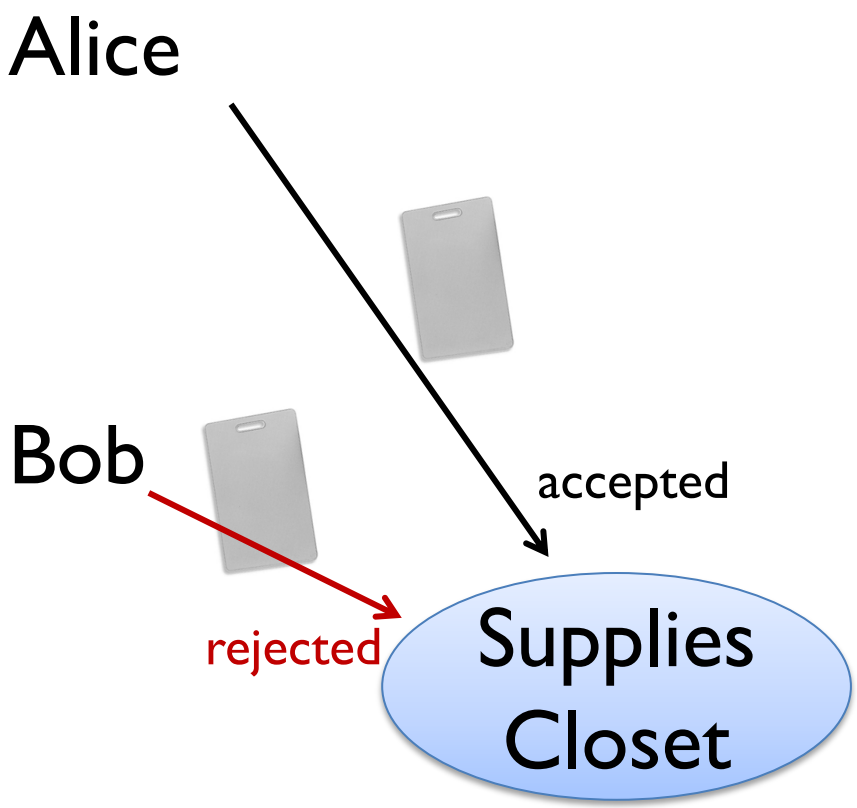
"Confused Deputy Problem"
e.g. cross-site request forgery (CSRF)

“Don’t prohibit what you can’t prevent”

Capabilities support decentralized delegation



Auth/RBAC systems are centralized



Bob has only been working there for 6 months ☹

Auth/RBAC systems are centralized

Alice

"Please grant Bob access to the supplies closet"
"Then revoke access after 20 minutes"

Can your system do this?

Authorization System

Auth/RBAC systems are centralized

Annoying and dangerous

Alice

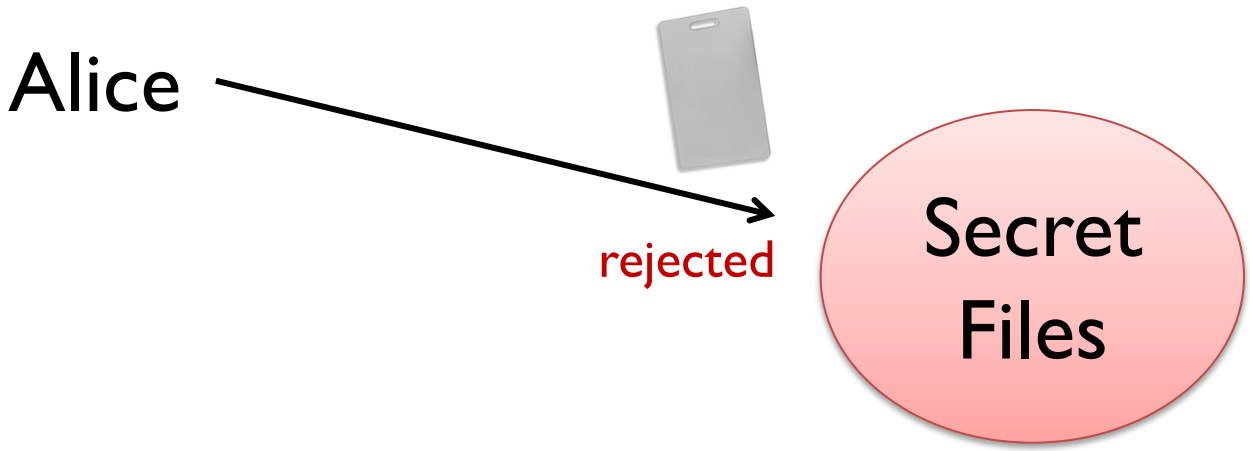
"Here, Bob. You can borrow my swipe card"

Bob

Supplies Closet

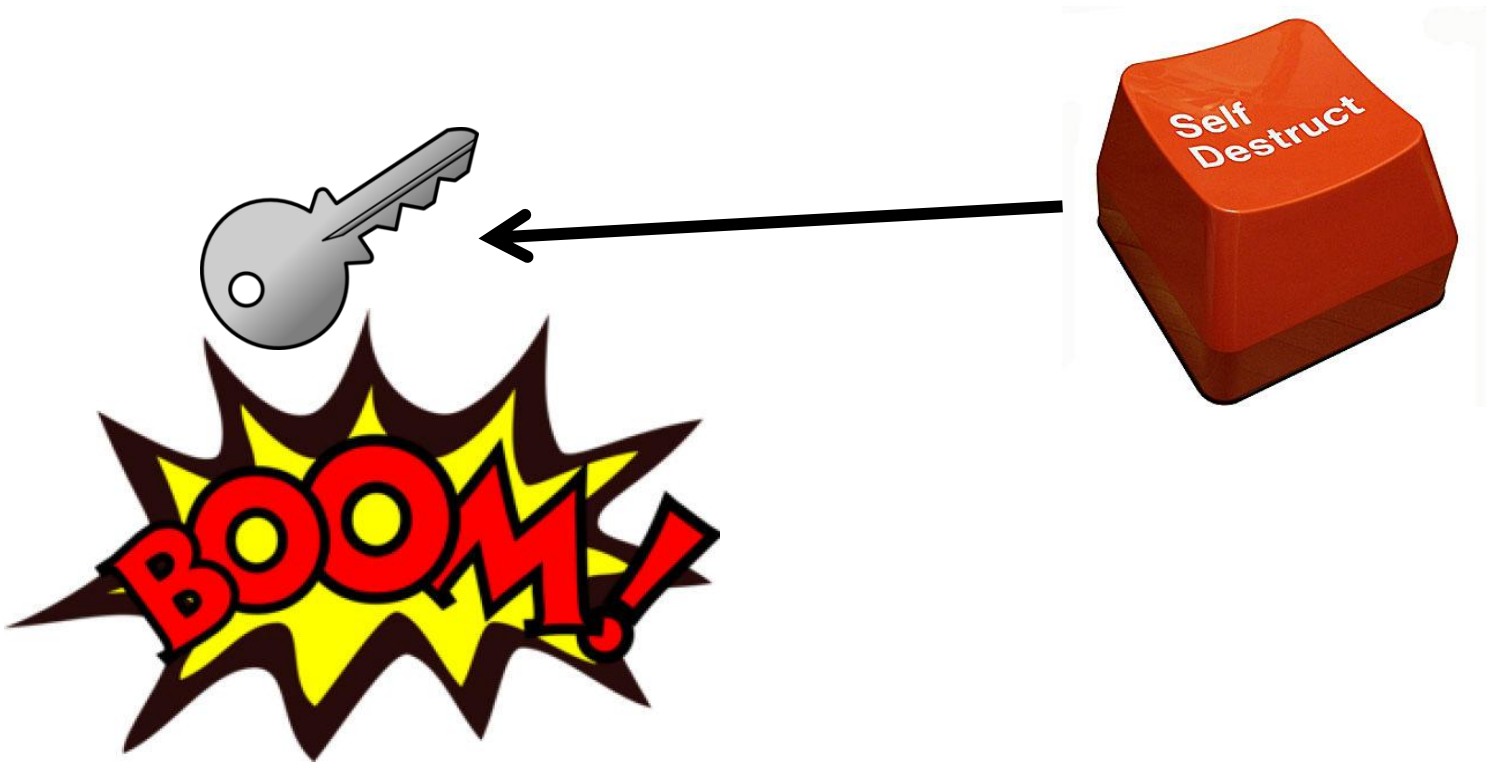
Credential Sharing

Central systems can revoke access

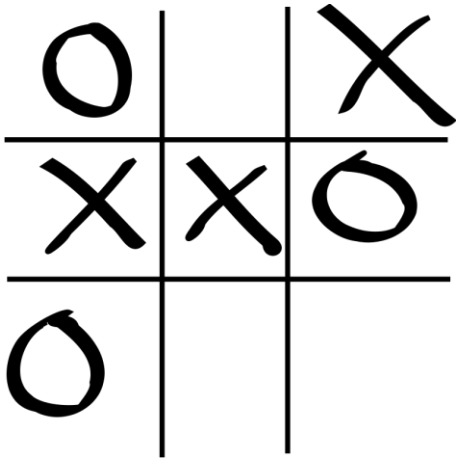


How to revoke access in a cap-based system?

How to revoke access in a cap-based system?



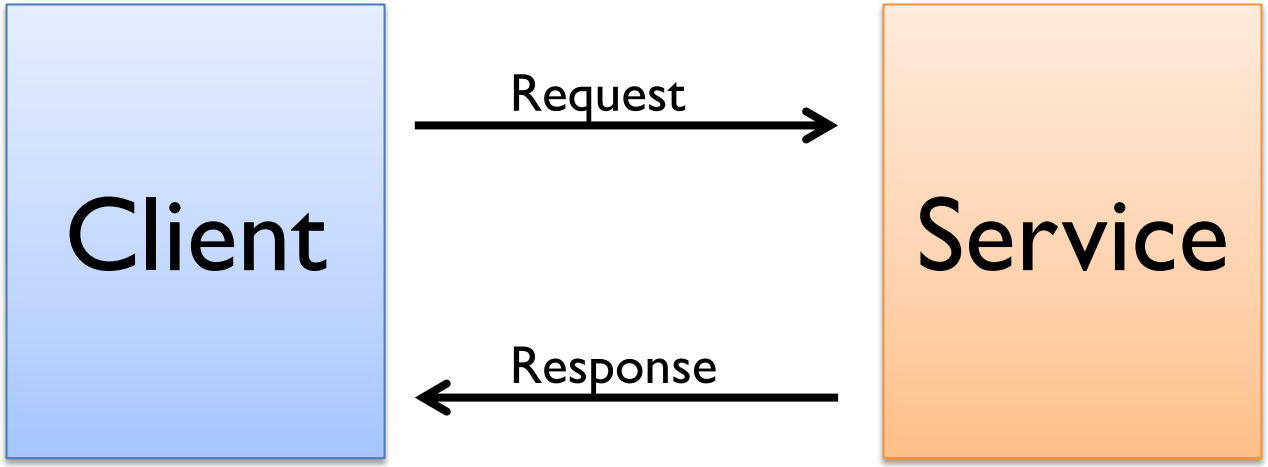
Demo: Capabilities as functions



DESIGNING AN API USING CAPABILITIES

Tic-Tac-Toe as a service

Proper name is "Noughts and Crosses" btw



Tic-Tac-Toe API (obvious version)

```
type TicTacToeRequest = {
    player: Player
    row: Row
    col: Column
    }
```

Tic-Tac-Toe API (obvious version)

```
type TicTacToeResponse = {
    result : MoveResult
    display: DisplayInfo
    }

type MoveResult =
    | KeepPlaying
    | GameWon of Player
    | GameTied
```

| means "a choice"

Demo: Tic-Tac-Toe

What kind of errors can happen?

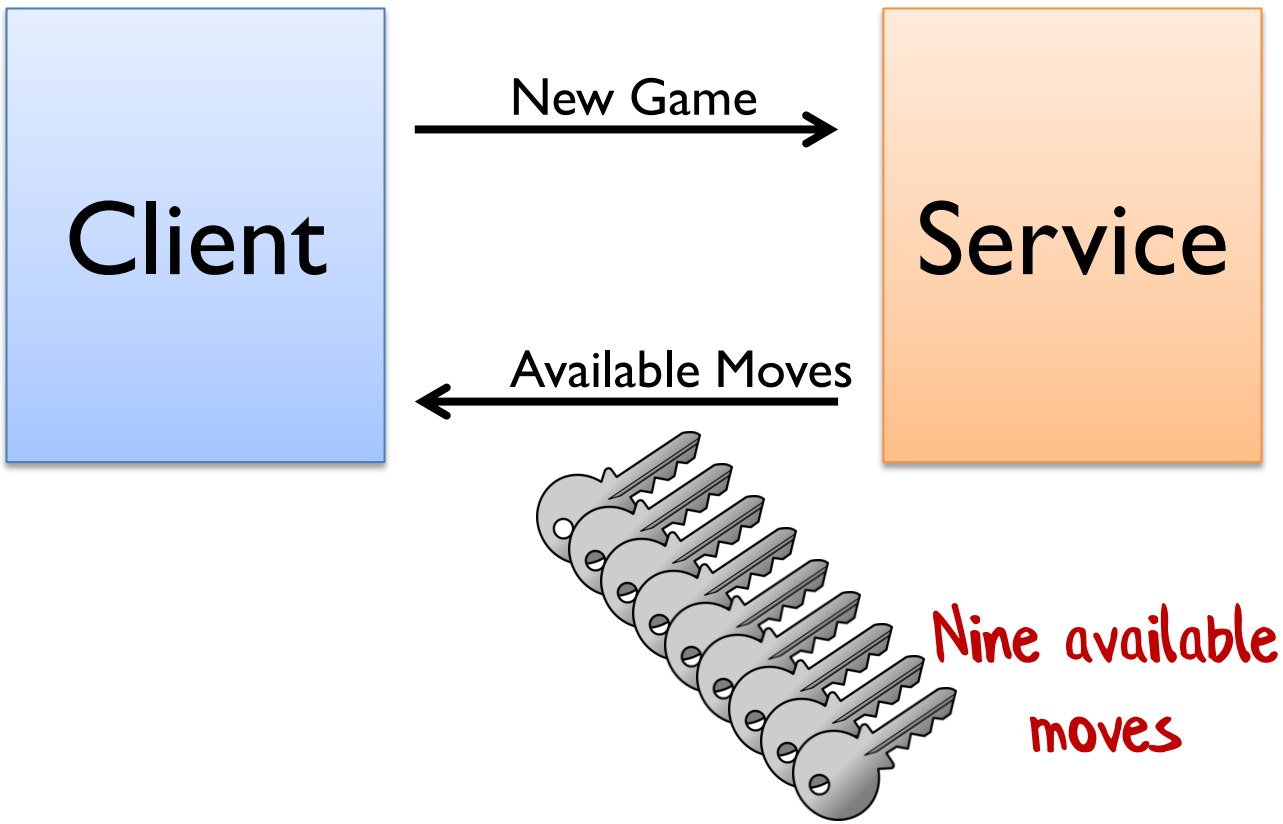
- A player can play an already played move
- A player can play twice in a row
- A player can forget to check the Result and keep playing

Yes, you could return errors, but...

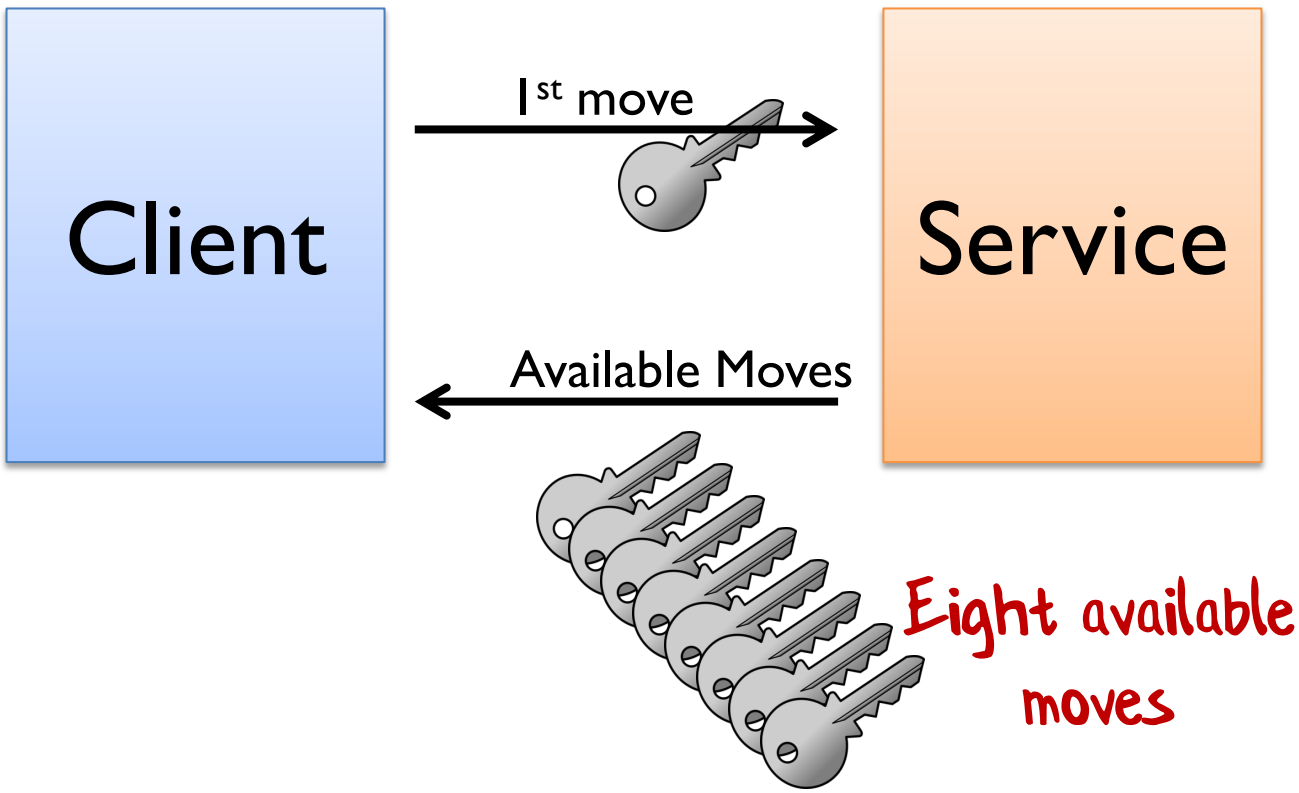
Don't let me do a bad thing and then tell me off for doing it...

"Make illegal operations unrepresentable"

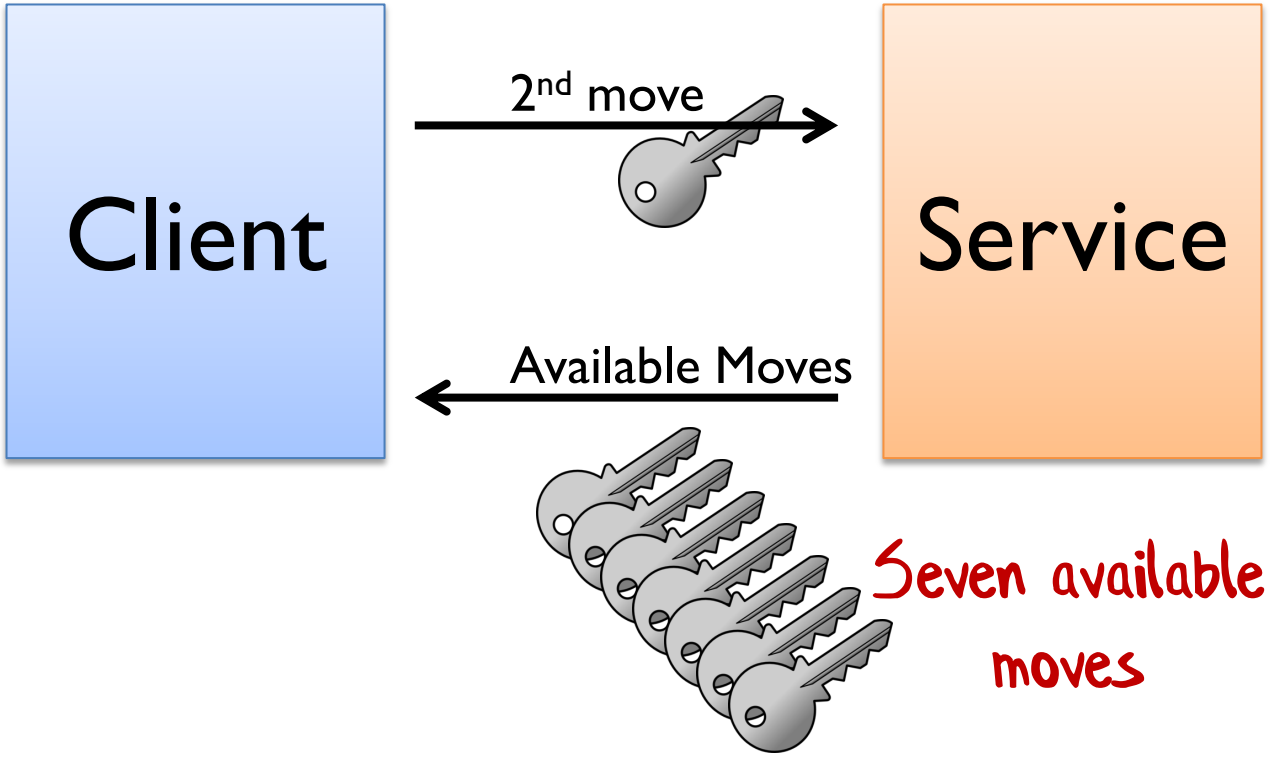
Tic-Tac-Toe service with caps



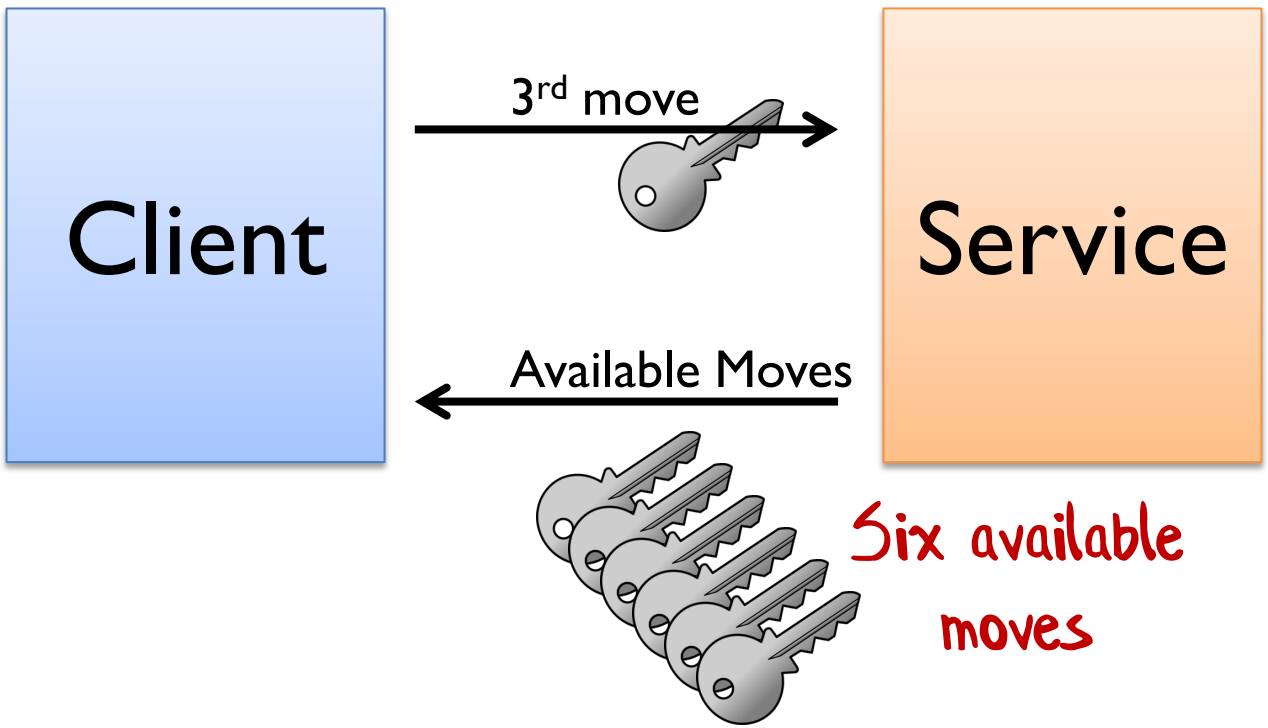
Tic-Tac-Toe service with caps



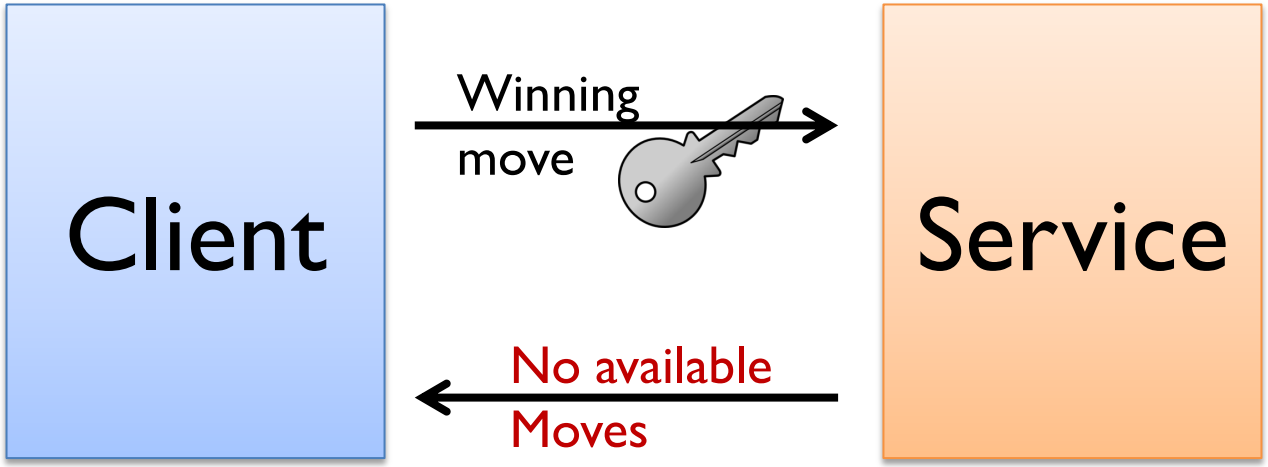
Tic-Tac-Toe service with caps



Tic-Tac-Toe service with caps



Tic-Tac-Toe service with caps



Tic-Tac-Toe API (cap-based version)

```
type MoveCapability =
    unit -> TicTacToeResponse

type NextMoveInfo = {
  playerToPlay : Player
  posToPlay : CellPosition
  capability : MoveCapability
  }
```

These are for client information only.

The player and position are baked into the capability

Tic-Tac-Toe API (cap-based version)

```
type TicTacToeResponse =
  | KeepPlaying of
      (DisplayInfo * NextMoveInfo list)
  | GameWon of
      (DisplayInfo * Player)
  | GameTied of DisplayInfo
```

* means "a pair"

Tic-Tac-Toe API (cap-based version)

```
type TicTacToeResponse =
  | KeepPlaying of
      (DisplayInfo * NextMoveInfo list)
  | GameWon of
      (DisplayInfo * Player)
  | GameTied of DisplayInfo
```

Response contains all available moves

Where did the "request" type go?

Where's the authorization?

Demo: Cap-based Api

What kind of errors can happen?

- ~~A player can play an already played move~~
- ~~A player can play twice in a row~~
- ~~A player can forget to check the Result and keep playing~~

All fixed now! ☺

Is this good security or good design?

RESTful done right

HATEOAS
Hypermedia As The Engine Of Application State

"A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia."

How NOT to do HATEOAS

```
POST /customers/
GET /customer/42
```

If you can guess the API
you're doing it wrong

Security problem!

Also, a design problem –
too much coupling.

How to do HATEOAS

```
POST /81f2300b618137d21d /
GET /da3f93e69b98
```

You can only know what URLs to use by parsing the page

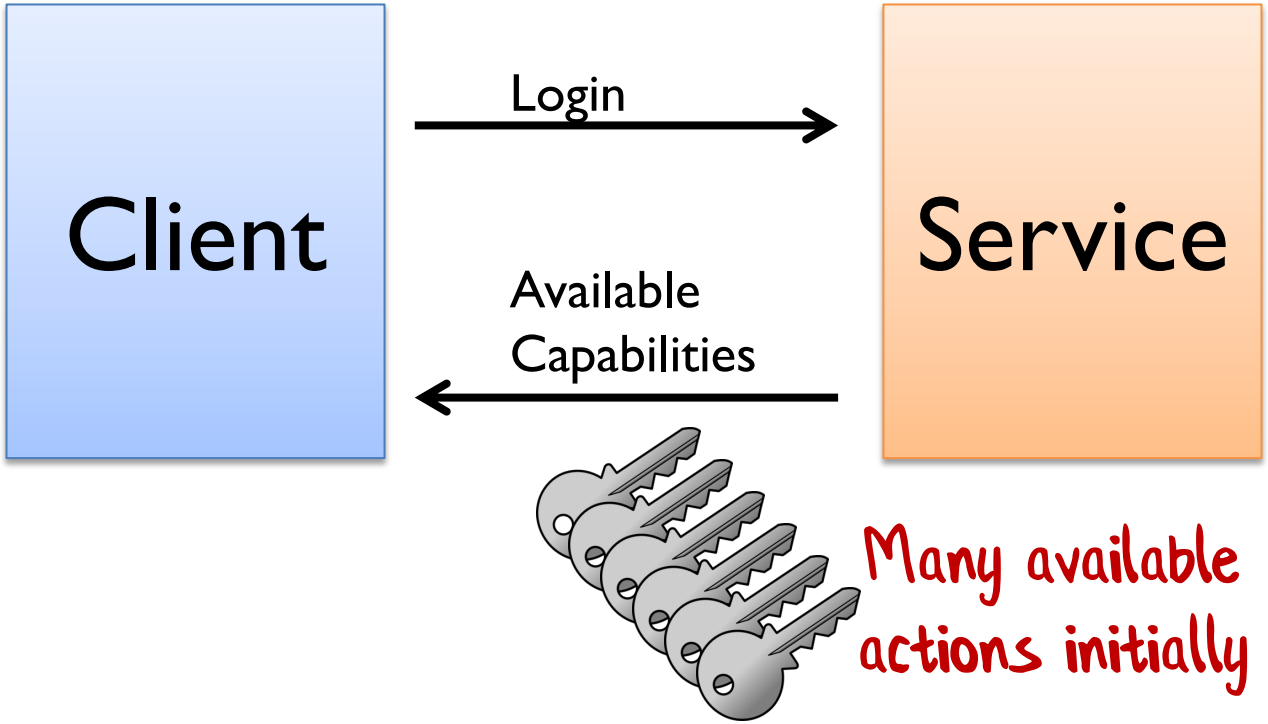
Each of these URLs is a capability

Demo: HATEOAS

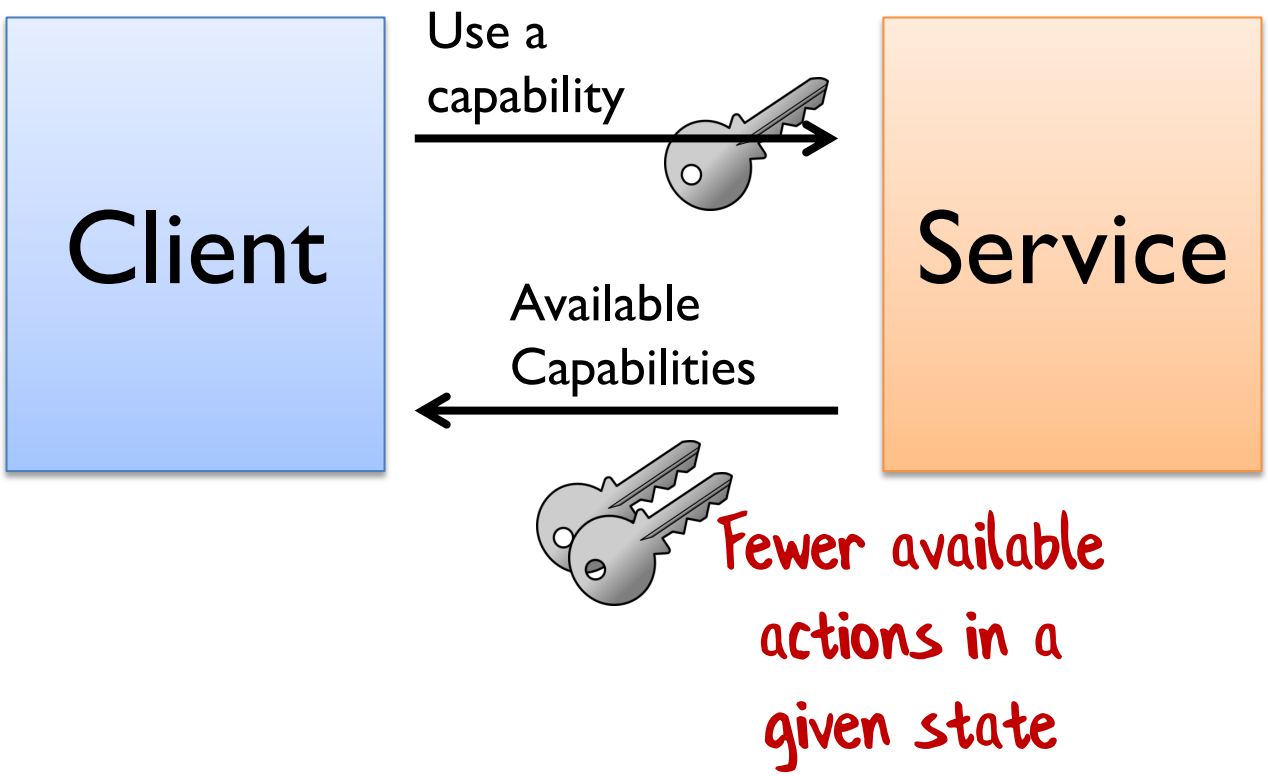
Some Benefits of HATEOAS

- Client decoupled from server
  - The server owns the API model and can change it without breaking any clients
  - E.g. Change links to point to CDN
  - E.g. Versioning
- Simpler clients in many cases
  - No need for client-side checking of moves
- Explorable API
  - Choose your own adventure!

Service API with caps

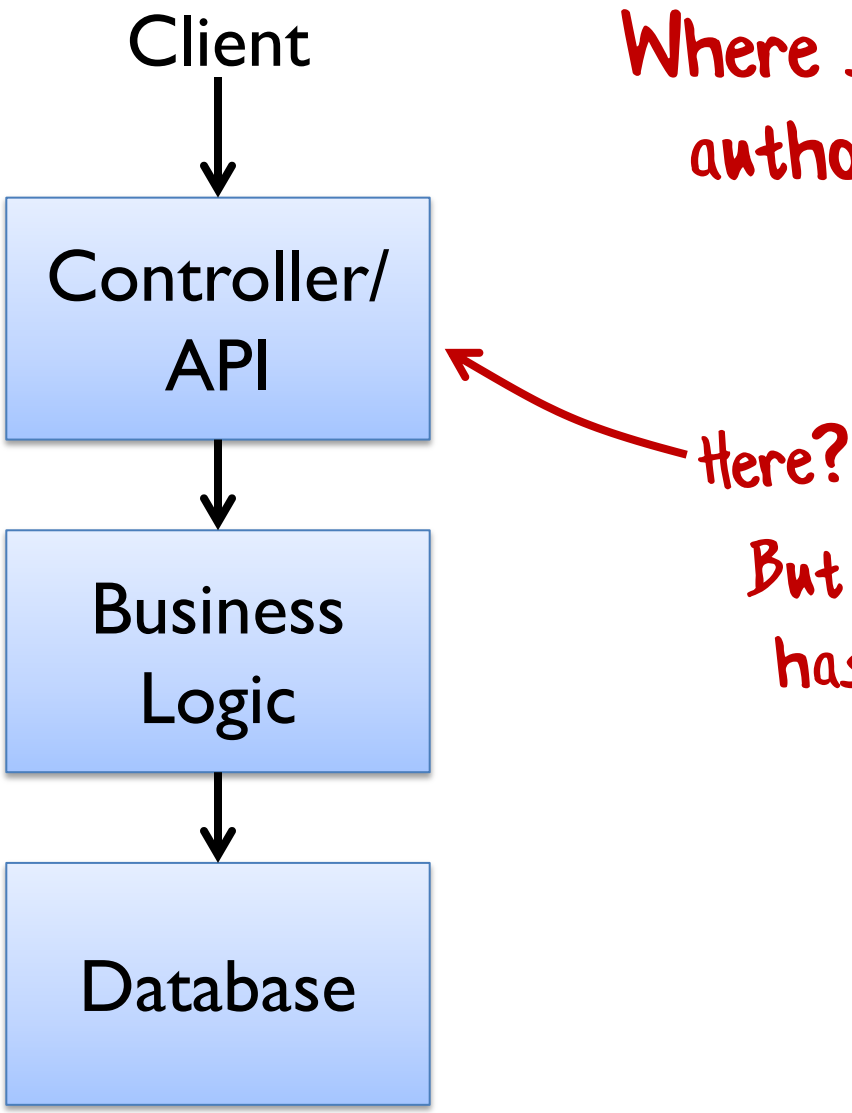


Service API with caps



CAPABILITY DRIVEN DESIGN

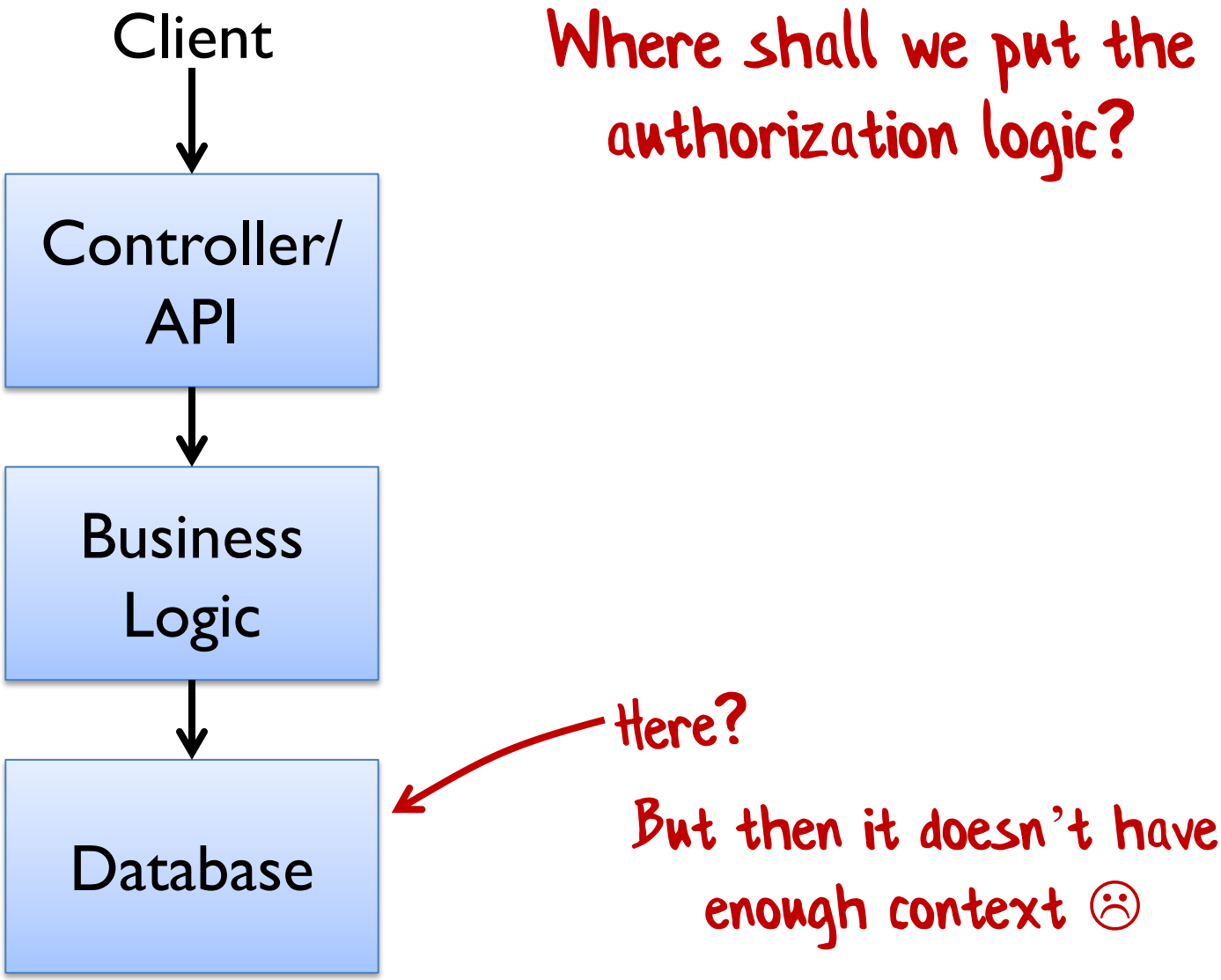
Using these design techniques throughout your domain

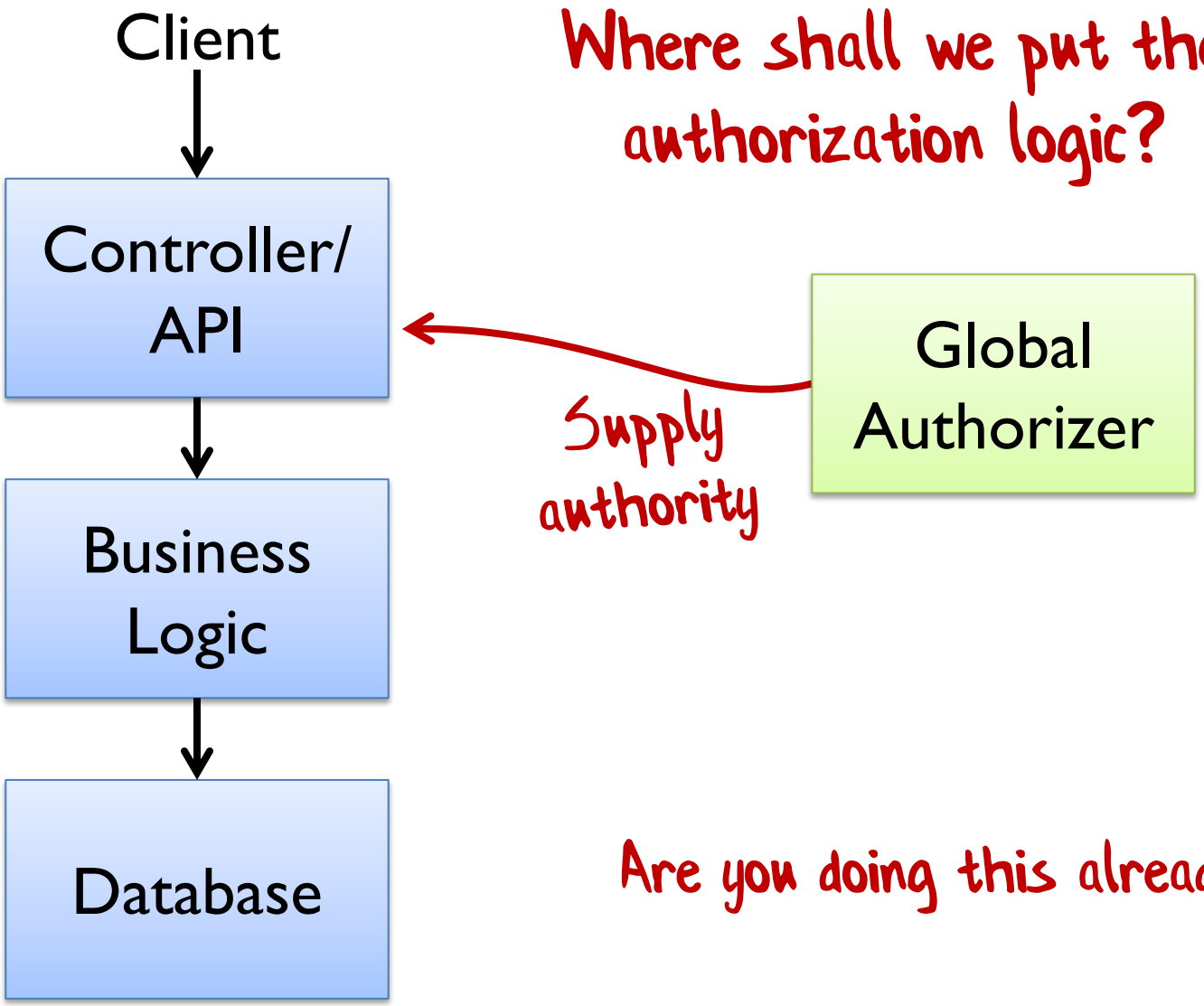


Where shall we put the authorization logic?

Here?

But then any other path has complete access to the database ☹


Client
Controller/ API
Business Logic
Database
Where shall we put the authorization logic?
Here?
But then it doesn't have enough context ☹



Where shall we put the authorization logic?

Are you doing this already?

```
public class CustomerController : ApiController
{
    readonly ICustomerDb _db;

    public CustomerController(ICustomerDb db)
    {
        _db = db;
    }

    [Route("customers/{customerId}")]
    [HttpGet]
    [GetCustomerProfileAuth]
    public IHttpActionResult Get(int customerId)
    {
        var custId = new CustomerId(customerId);
        var cust = _db.GetProfile(custId);
        var dto = DtoConverter.CustomerToDto(cust);
        return Ok(dto);
    }
```

Inject authority to access db

Custom auth attribute

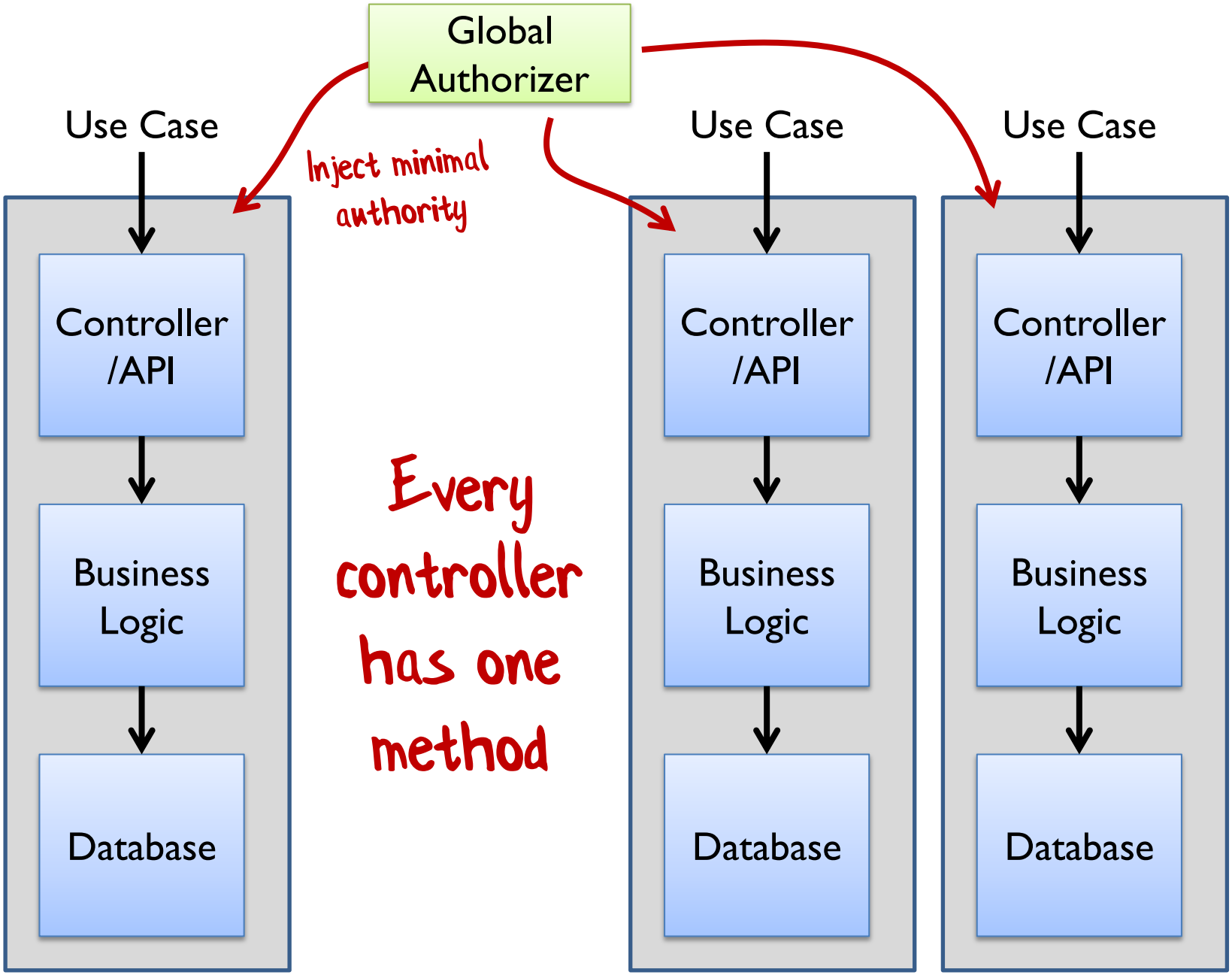
Use the authority

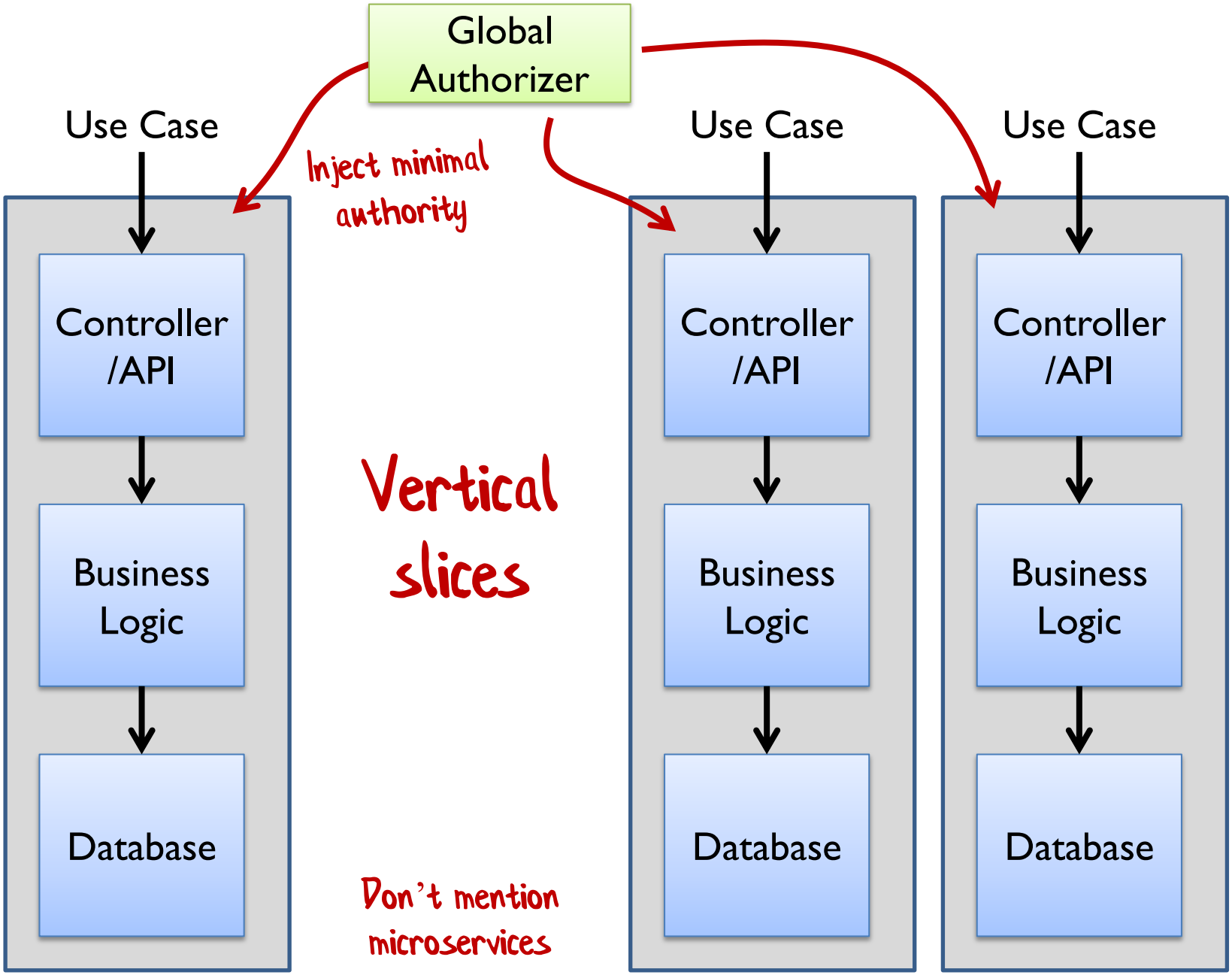
How much authority do you really need?

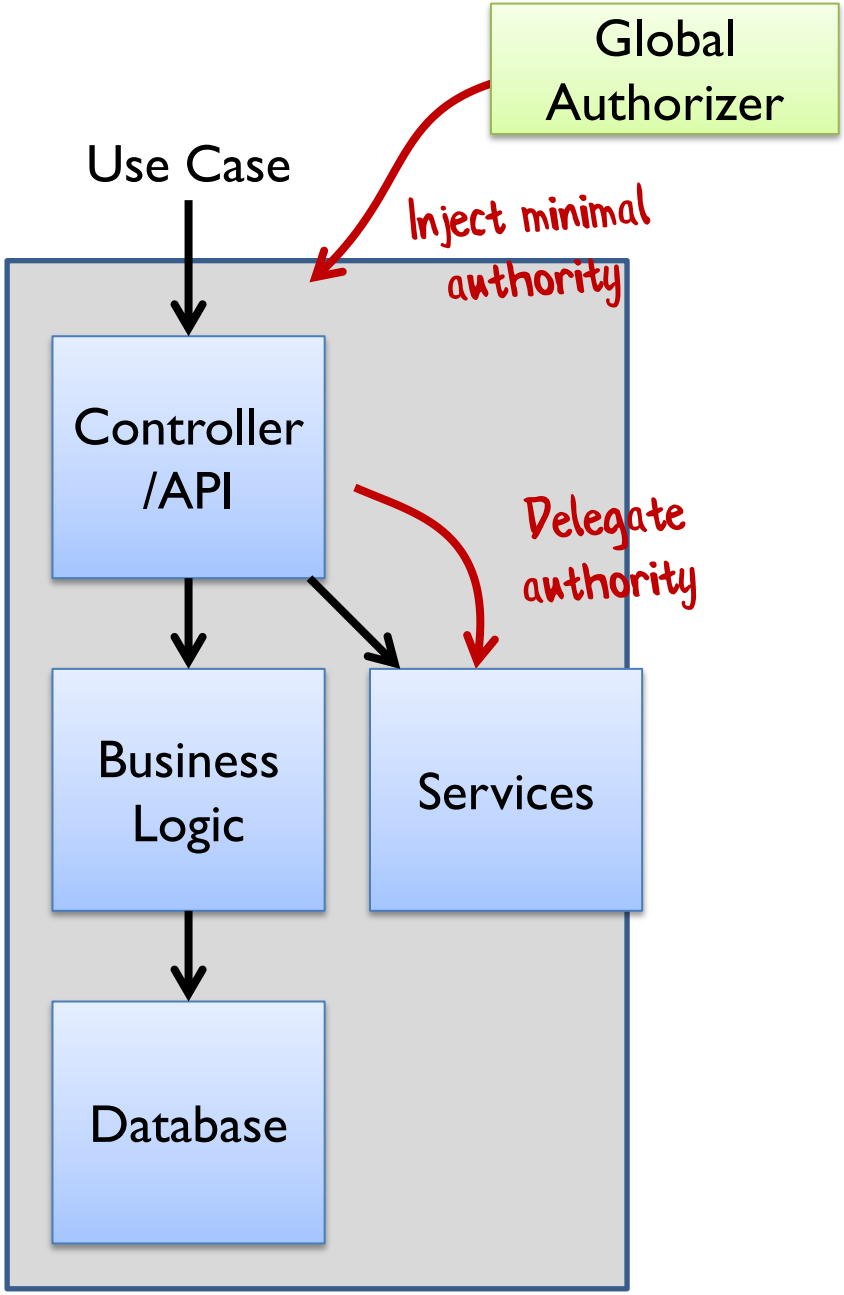
```
public interface ICustomerDb
{
  CustomerProfile GetProfile(CustomerId id);
  void UpdateProfile(CustomerId id, CustomerProfile cust);

  void CreateAccount(CustomerId id, CustomerProfile cust);
  void DeleteAccount(CustomerId id);

  void UpdateLoginEmail(CustomerId id, string email);
  void UpdatePassword(CustomerId id, string password);
}
```

Too much authority!

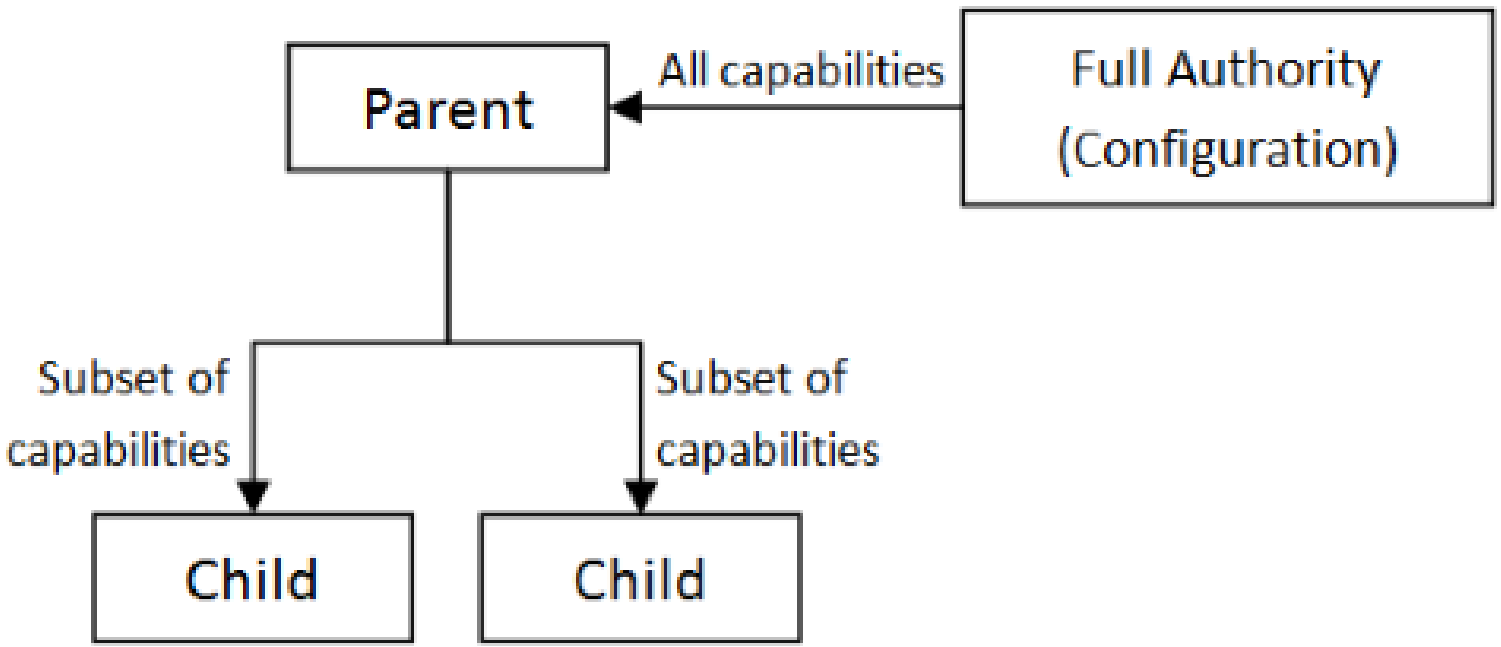
The only authority I need


Global Authorizer
Inject minimal authority
Use Case
Controller /API
Business Logic
Database
Use Case
Controller /API
Business Logic
Database
Use Case
Controller /API
Business Logic
Database
Every controller has one method

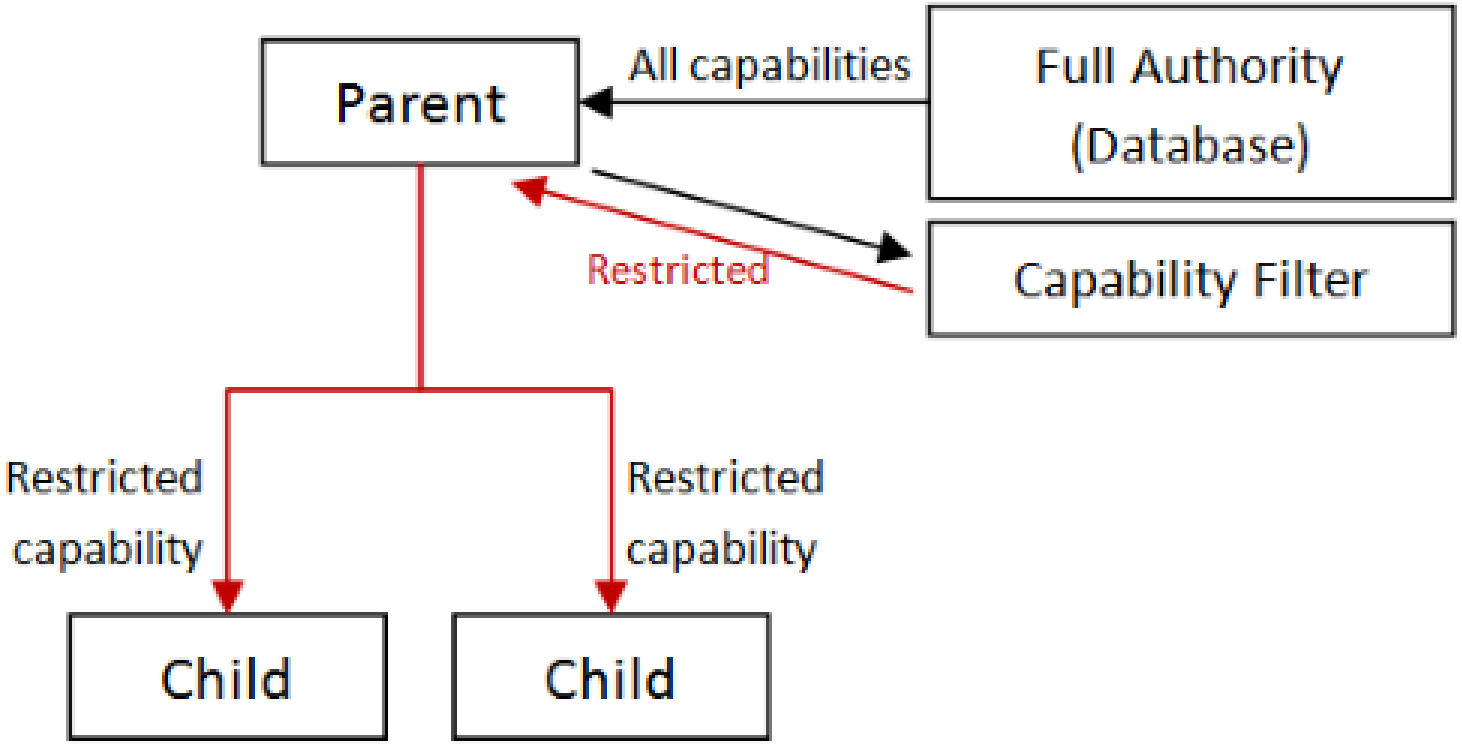

Global Authorizer
Use Case
Use Case
Use Case
Inject minimal authority
Controller /API
Controller /API
Controller /API
Vertical slices
Business Logic
Business Logic
Business Logic
Database
Database
Database
Don't mention microservices


Global Authorizer
Use Case
Inject minimal authority
Controller /API
Delegate authority
Business Logic
Services
Database

Delegation of capabilities



Delegation with restriction



```
type MessageFlag =
 ShowThisMessageAgain | DontShowThisMessageAgain

type ConfigurationCapabilities = {
  GetMessageFlag : unit -> MessageFlag
  SetMessageFlag : MessageFlag -> unit
  GetBackgroundColor : unit -> Color
  SetBackgroundColor : Color -> unit
  GetConnectionString : unit -> ConnectionString
  SetConnectionString : ConnectionString -> unit
  }
```

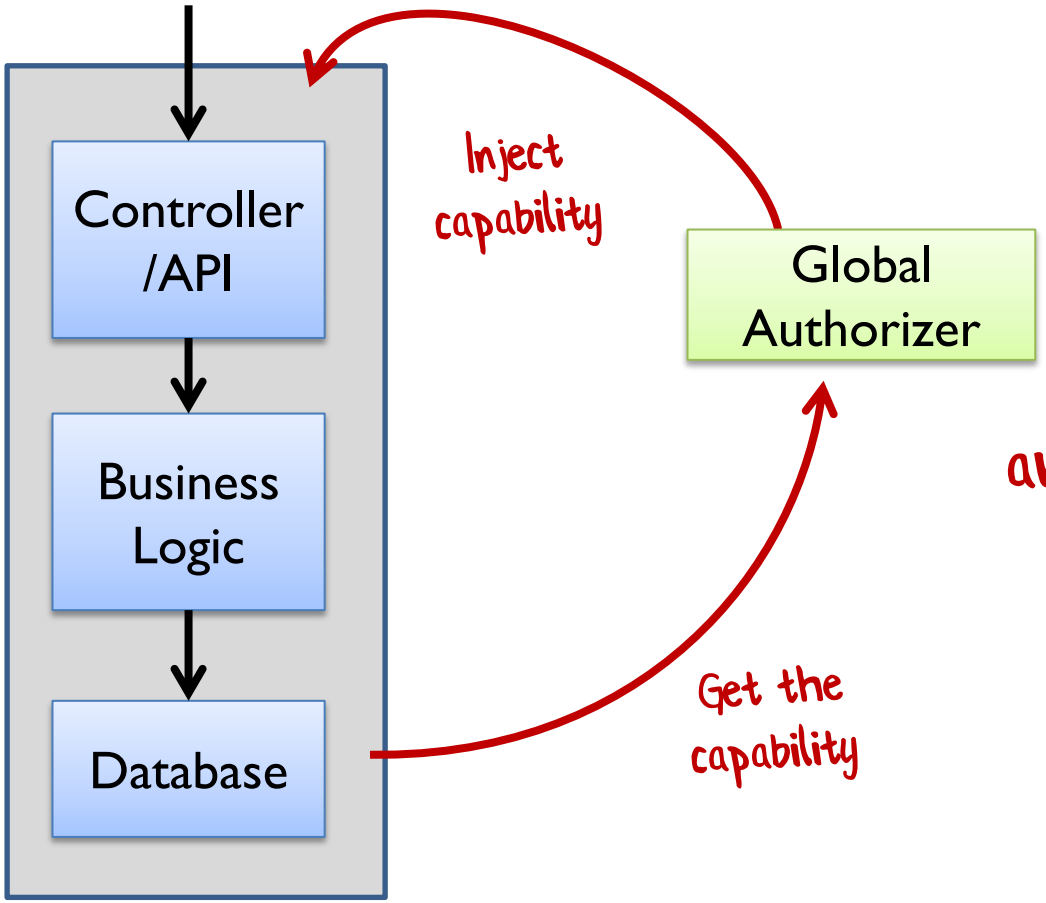
Delegate this to dialog box

```
let dontShowMessageAgainDialogBox capabilities =
  let getFlag,setFlag = capabilities
  let ctrl= new CheckBox(
    Text="Don't show this message again")
  ctrl.Checked <- getFlag()
  // etc
```

The only access to the outside world

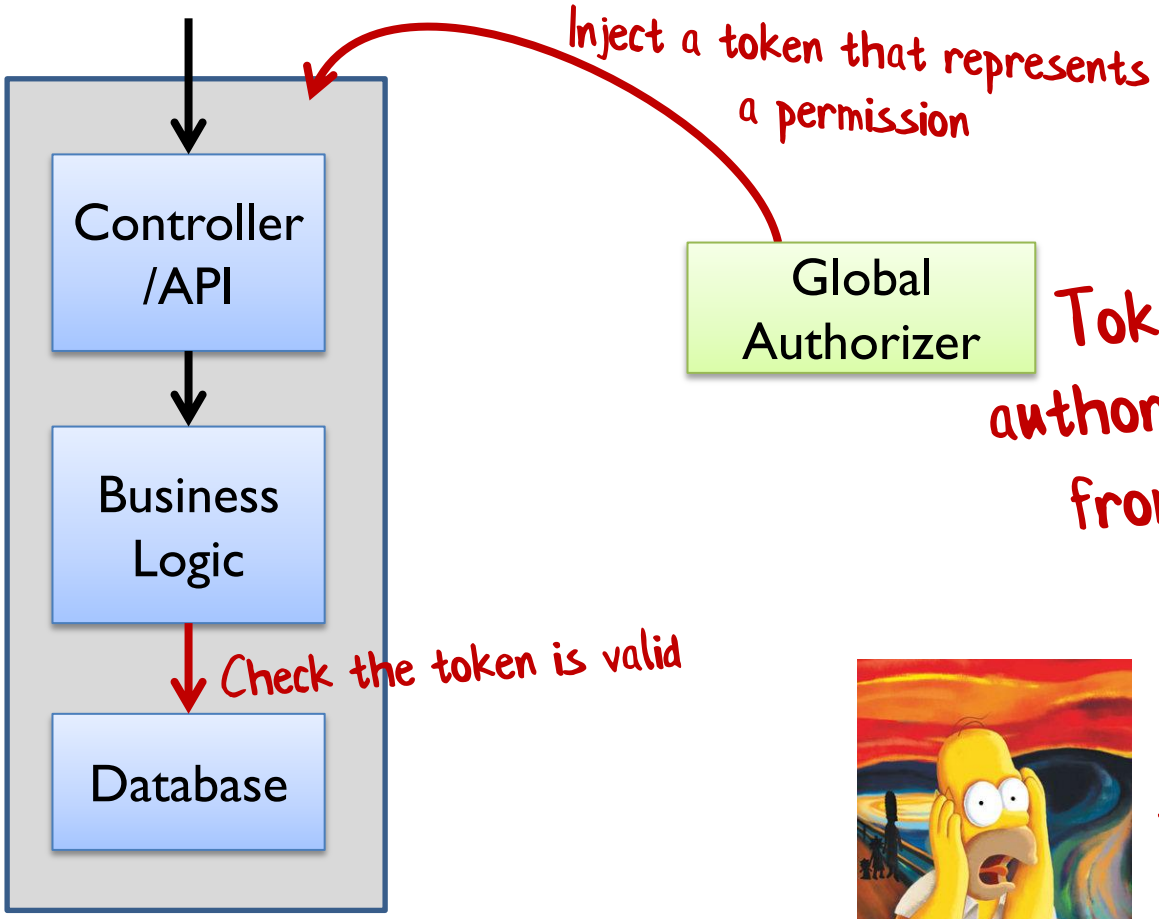
Demo:
Vertical slices and delegation

Passing capabilities



Problem: global authorizer needs access to the database

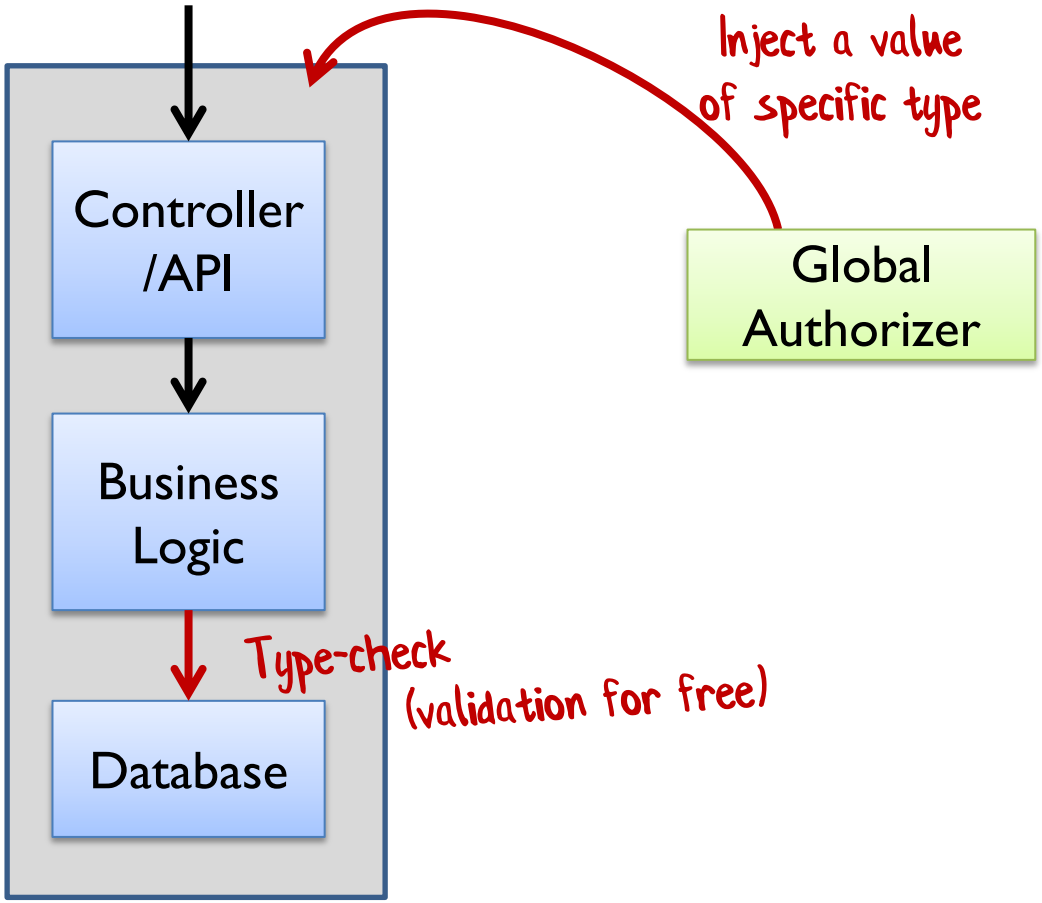
Passing tokens



Tokens let global authorizer be decoupled from database ☺

Overkill?
Too complicated to implement?

Passing tokens as types



Represent tokens by types!

Demo:
Using types for access tokens

What have we covered?

- **Don't** use complicated security patterns
  - This will ensure that they are never used, or used wrong.
  - Don't rely on other people doing the right thing.
  - Don't rely on other people reading the documentation!
- **Do** use techniques where you get security for free.
  - You can be lazy!
  - You don't have to remember to do anything.

What have we covered?

- **Don't** develop first, add security later
  - Right after you implement the "quality" module
- **Do** use security-driven design
  - Bonus: get a modular architecture!

What have we covered?

- **Don't** only do security at the outermost layer
- **Do** use POLA everywhere, which ensures that you have minimal dependencies.

Common questions

- How do you pass these capabilities around?
  - Dependency injection or equivalent
- Are you saying that *all* external IO should be passed around as capabilities?
  - Yes! You should never access any ambient authority.
  - You should be doing this anyway for mocking.

Common questions

- Won't there be too many parameters?
  - Less than you think!
  - Encourages vertical slices (per use-case, scenario)
  - "Functional core, imperative shell"
- Can't this be bypassed by reflection or other backdoors?
  - Yes. This is really all about design not about total security.

Resources for capability-based thinking

- LMGTFY "Capability based security"
- "Lazy developers guide to secure computing" talk by Marc Stiegler
- erights.org
- Google's Caja built over JavaScript
- Emily, a capability based language (via Ocaml)

Thanks!

@ScottWlaschin

Contact me

fsharpforfunandprofit.com/cap

Slides and video here

Let us know if you need help with F#

fsharpWorks

More F# at fsharp.org