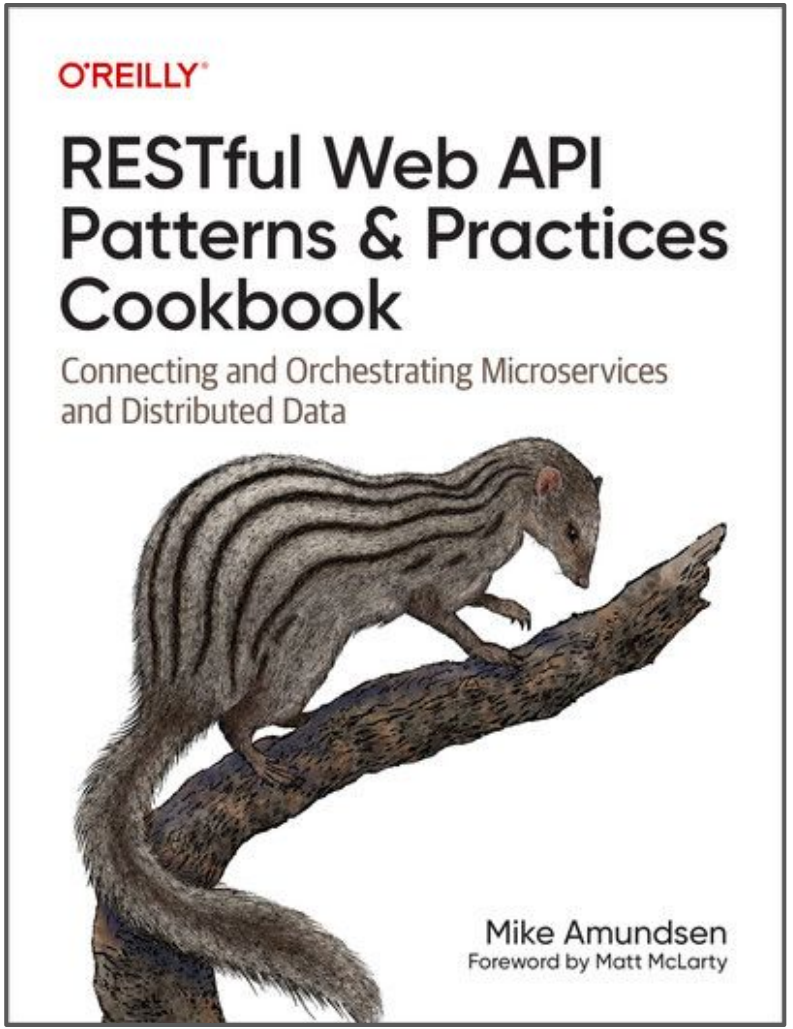
RESTful Web API Patterns and Practices

Mike Amundsen
@mamund


Mike Amundsen
@mamund

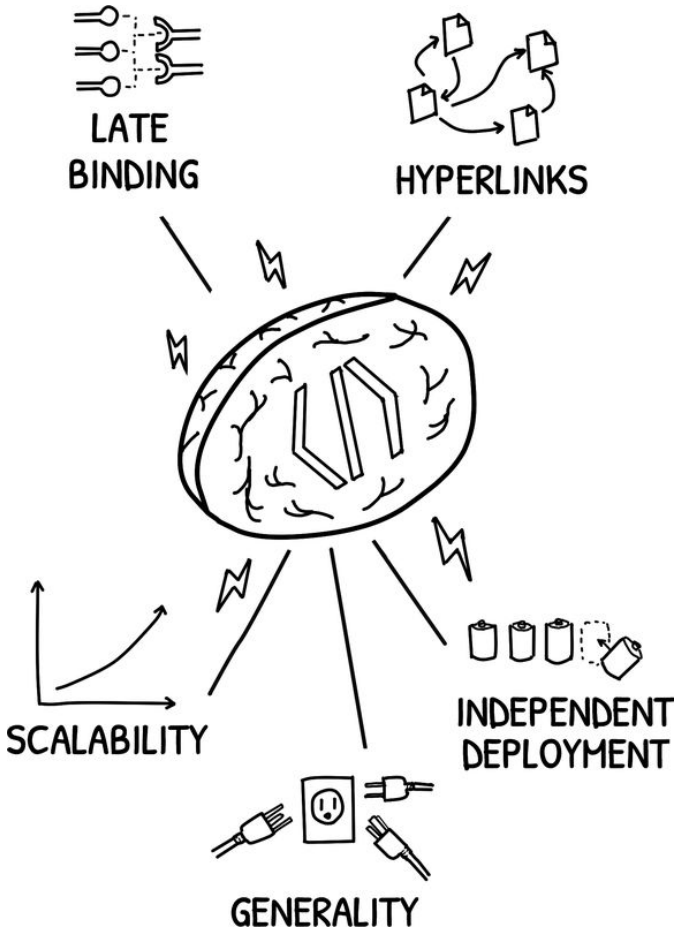
"This guide shows you the rules, routines, commands, and protocols—the glue—that integrates individual microservices so they can function together in a safe, scalable, and reliable way."

-- O'Reilly Media



Overview

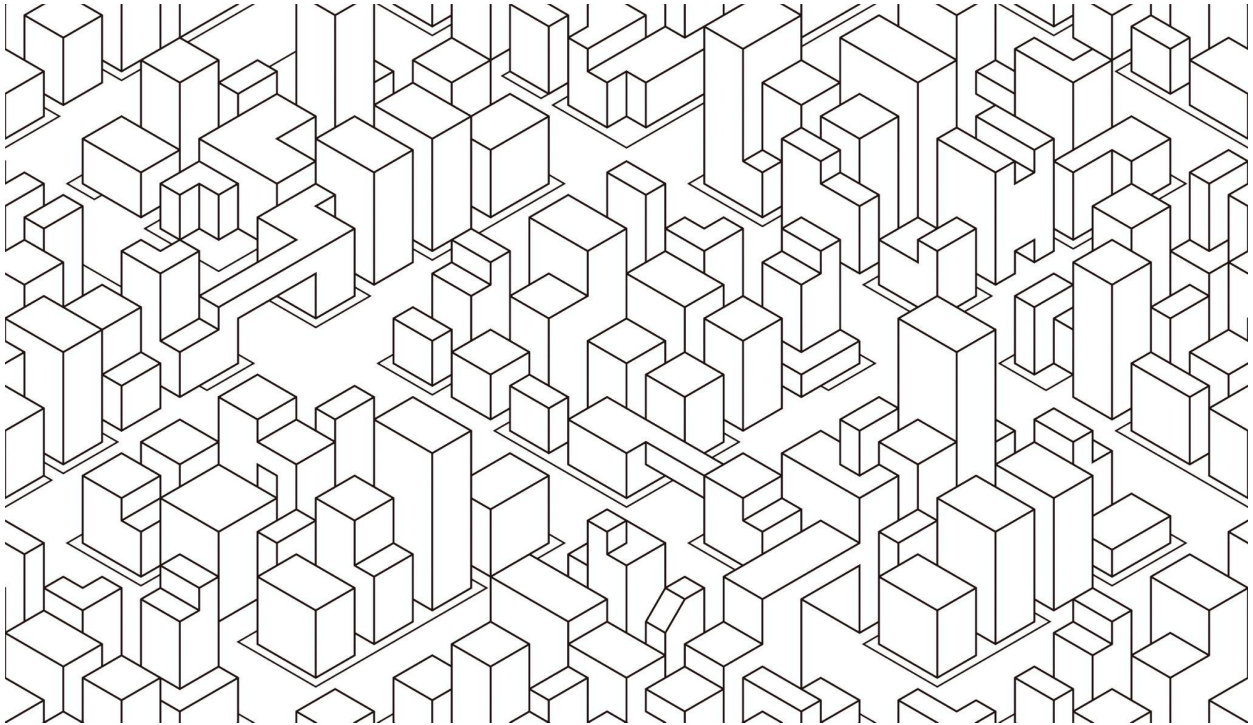
- Pattern Thinking
- Design
- Clients
- Services
- Data
- Workflow
- Summary



Pattern Thinking

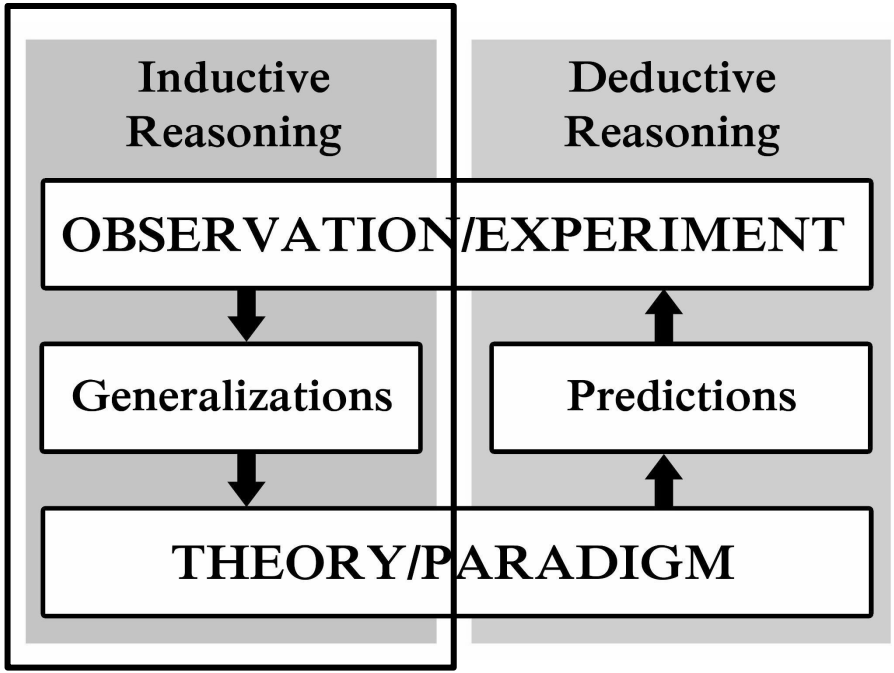
Pattern Thinking

A framework for understanding, designing, and constructing systems



Pattern Thinking

Inductive reasoning is any of various methods of reasoning in which broad generalizations or principles are derived from a body of observations.



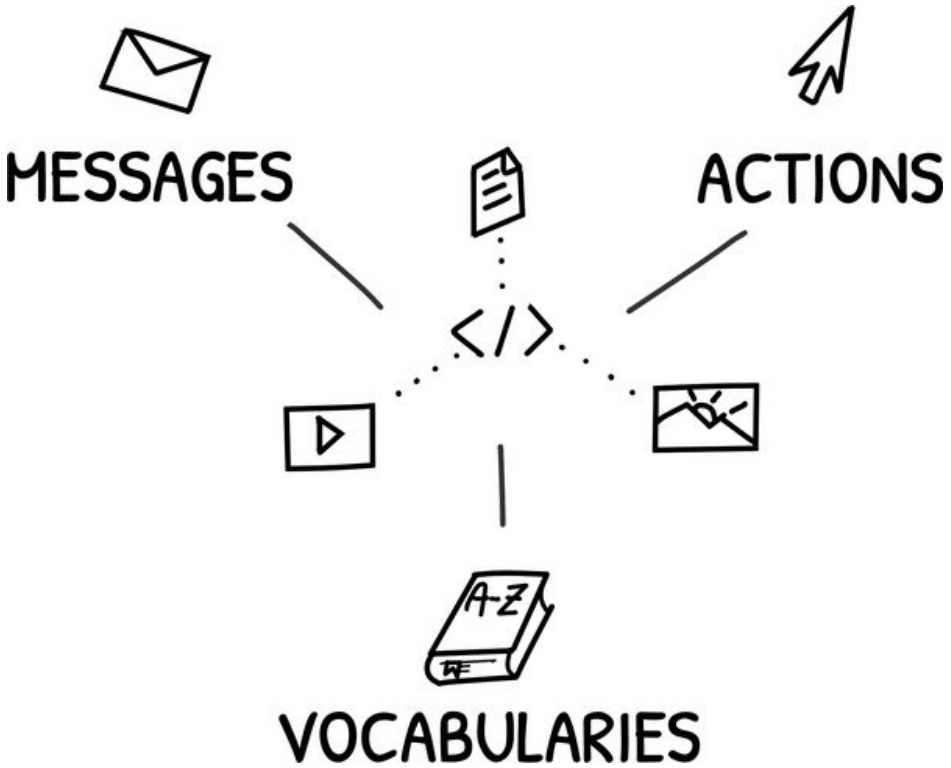
Pattern Thinking

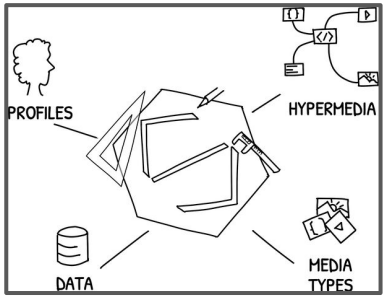
"Each pattern describes a problem which occurs over and over again, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

-- Christopher Alexander

Pattern Thinking

Web-centric implementations rely on three key elements: messages, actions, and vocabularies.



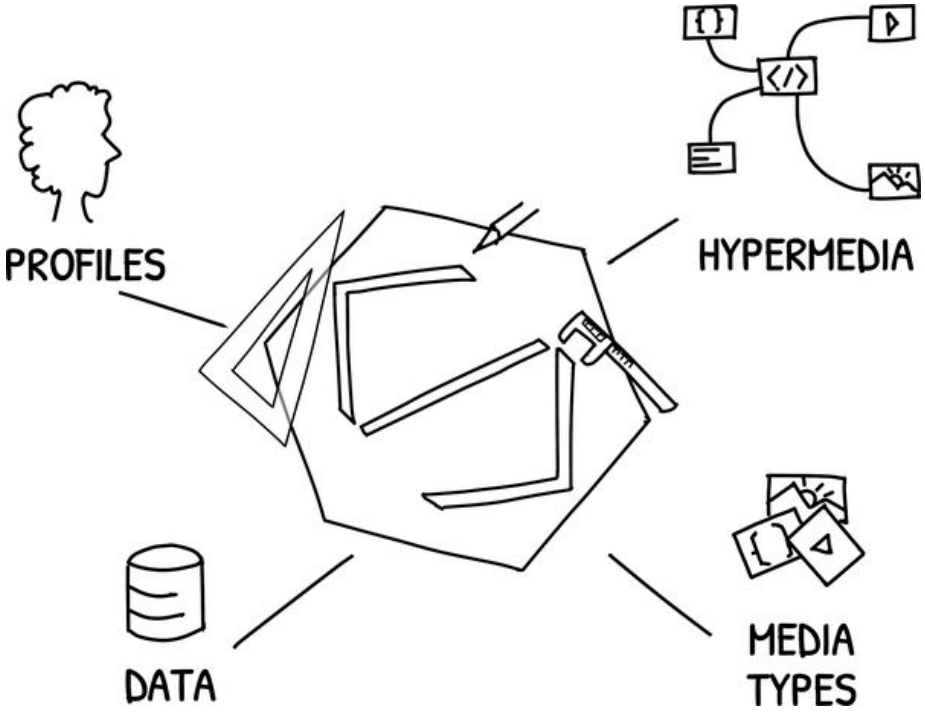


Design

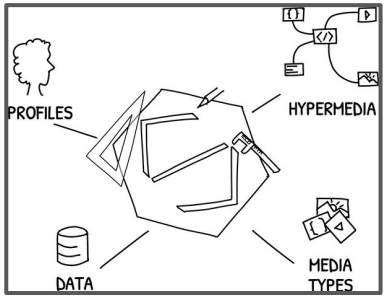
> The problem is essentially the one discussed by science fiction writers: "how do you get communications started among totally uncorrelated 'sapient' beings?"
>
> —J.C.R. Licklider, *1966*

Design Patterns

*Design systems so that **machines** built by different **people** who have never met can successfully **interact** with each other.*

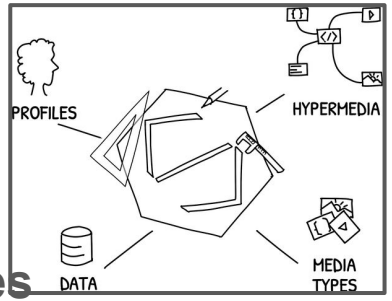


Design Patterns



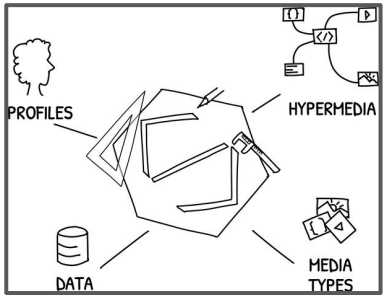
3.1 Creating Interoperability with Registered Media Types
3.2 Ensuring Future Compatibility with Structured Media Types
3.3 Sharing Domain Specifics Via Published Vocabularies
3.4 Describing Problem Spaces with Semantic Profiles
3.5 Expressing Domain Actions at Run-time with Embedded Hypermedia
3.6 Designing Consistent Data Writes with Idempotent Actions
3.7 Enabling Interoperability with Inter-Service State Transfers
3.8 Design for Repeatable Actions
3.9 Design for Reversible Actions
3.10 Design for Extensible Messages
3.11 Design for Modifiable Interfaces

Design Patterns



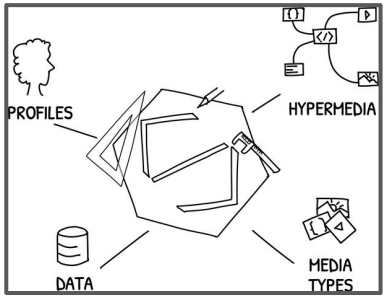
3.1 Creating Interoperability with Registered Media Types
3.2 Ensuring Future Compatibility with Structured Media Types
3.3 Sharing Domain Specifics Via Published Vocabularies
3.4 Describing Problem Spaces with Semantic Profiles
3.5 Expressing Domain Actions at Run-time with Embedded Hypermedia
3.6 Designing Consistent Data Writes with Idempotent Actions
3.7 Enabling Interoperability with Inter-Service State Transfers
3.8 Design for Repeatable Actions
3.9 Design for Reversible Actions
3.10 Design for Extensible Messages
3.11 Design for Modifiable Interfaces

Design Patterns



3.1 Creating Interoperability with Registered Media Types
3.2 Ensuring Future Compatibility with Structured Media Types
3.3 Sharing Domain Specifics Via Published Vocabularies
3.4 Describing Problem Spaces with Semantic Profiles
3.5 Expressing Domain Actions at Run-time with Embedded Hypermedia
3.6 Designing Consistent Data Writes with Idempotent Actions
3.7 Enabling Interoperability with Inter-Service State Transfers
3.8 Design for Repeatable Actions
3.9 Design for Reversible Actions
3.10 Design for Extensible Messages
3.11 Design for Modifiable Interfaces

Design Patterns



3.1 Creating Interoperability with Registered Media Types
3.2 Ensuring Future Compatibility with Structured Media Types
3.3 Sharing Domain Specifics Via Published Vocabularies
3.4 Describing Problem Spaces with Semantic Profiles
3.5 Expressing Domain Actions at Run-time with Embedded Hypermedia
3.6 Designing Consistent Data Writes with Idempotent Actions
3.7 Enabling Interoperability with Inter-Service State Transfers
3.8 Design for Repeatable Actions
3.9 Design for Reversible Actions
3.10 Design for Extensible Messages
3.11 Design for Modifiable Interfaces

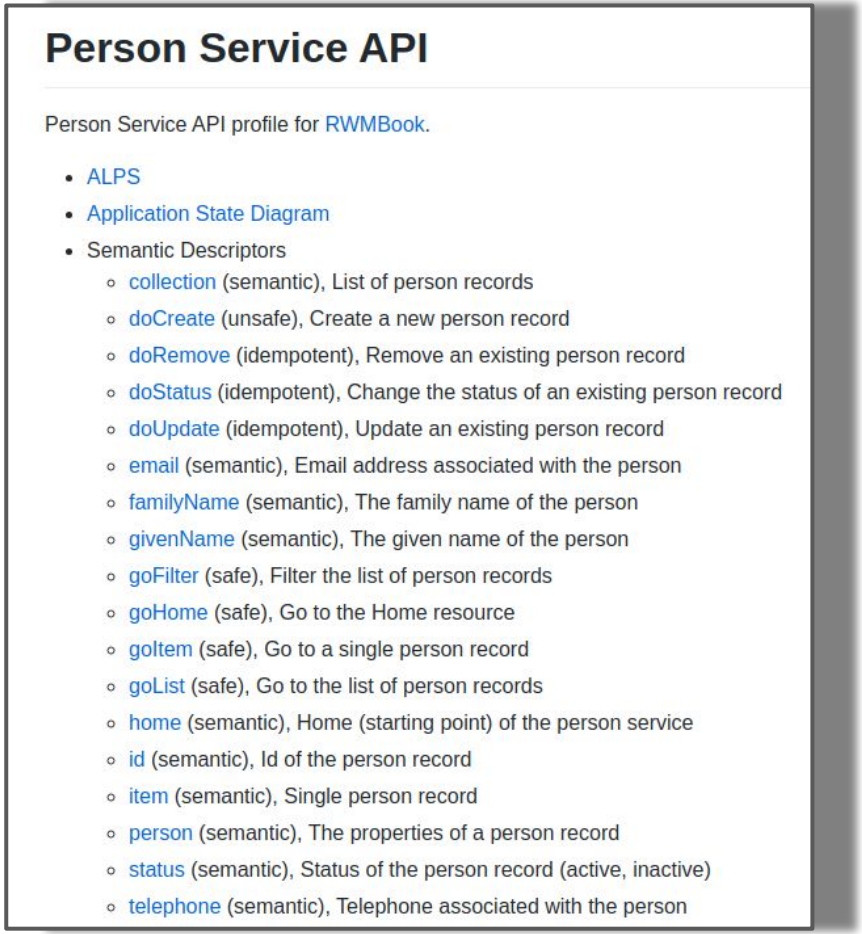
Design Patterns

Describing Problem Spaces with Semantic Profiles


```
{
   $schema: "https://alps-io.github.io/schemas/alps.json",
 - alps: {
      version: "1.0",
      title: "Person Service API",
    + doc: { … },
    - descriptor: [
       - {
            id: "id",
            type: "semantic",
            def: "https://schema.org/identifier",
            title: "Id of the person record",
            tag: "ontology",
          + doc: { … }
         },
       - {
            id: "givenName",
            type: "semantic",
            def: "https://schema.org/givenName",
            title: "The given name of the person",
            tag: "ontology",
          + doc: { … }
         },
       - {
            id: "familyName",
            type: "semantic",
            def: "https://schema.org/familyName",
            title: "The family name of the person",
            tag: "ontology",
          + doc: { … }
         },
       - {
```

Design Patterns

Describing Problem Spaces with Semantic Profiles



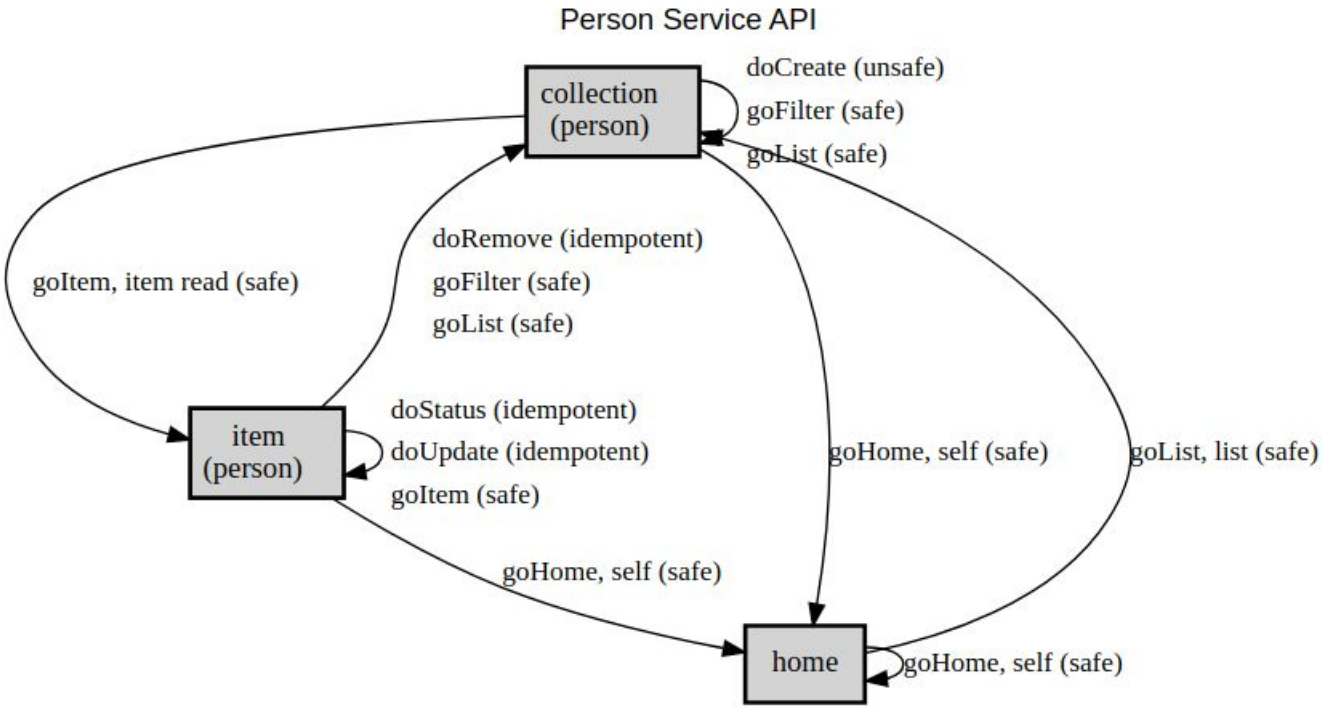
Person Service API

Person Service API profile for RWMBook.

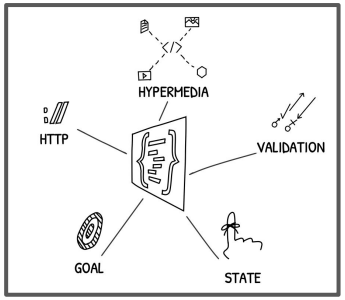
- ALPS
- Application State Diagram
- Semantic Descriptors
  - collection (semantic), List of person records
  - doCreate (unsafe), Create a new person record
  - doRemove (idempotent), Remove an existing person record
  - doStatus (idempotent), Change the status of an existing person record
  - doUpdate (idempotent), Update an existing person record
  - email (semantic), Email address associated with the person
  - familyName (semantic), The family name of the person
  - givenName (semantic), The given name of the person
  - goFilter (safe), Filter the list of person records
  - goHome (safe), Go to the Home resource
  - goItem (safe), Go to a single person record
  - goList (safe), Go to the list of person records
  - home (semantic), Home (starting point) of the person service
  - id (semantic), Id of the person record
  - item (semantic), Single person record
  - person (semantic), The properties of a person record
  - status (semantic), Status of the person record (active, inactive)
  - telephone (semantic), Telephone associated with the person

Design Patterns

Describing Problem Spaces with Semantic Profiles



Make designs composable

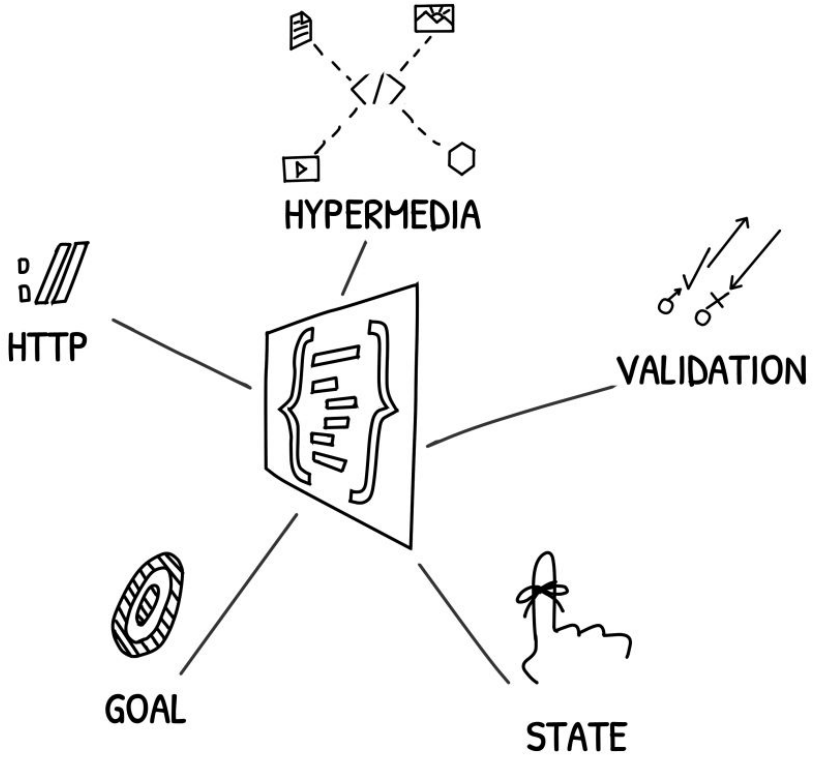


Clients

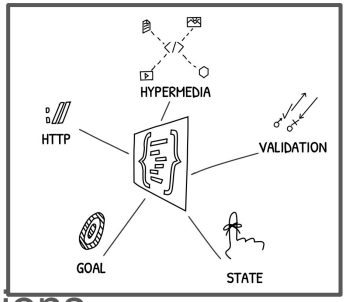
> The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.
>
> —Ted Nelson

Client Patterns

*Create API consumer apps that make few assertions about **how** they communicate (protocol, message model, and vocabulary) with servers and let the server supply the details (the **what**) at runtime.*

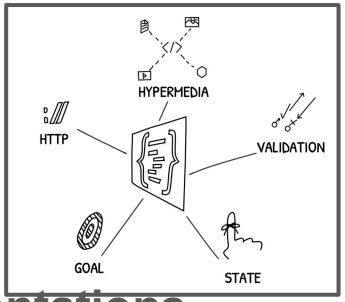


Client Patterns



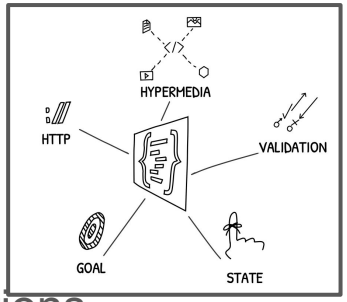
4.1 Limiting the use of Hard-Coded URLs
4.2 Code Clients to be HTTP-Aware
4.3 Coding More Resilient Clients With Message-Centric Implementations
4.4 Coding Effective Clients to Understand Vocabulary Profiles
4.5 Negotiate for Profile Support at Runtime
4.6 Managing Representation Formats At Runtime
4.7 Using Schema Documents as a Source of Message Metadata
4.8 Every Important Element Within a Response Needs an Identifier
4.9 Relying on Hypermedia Controls In the Response
4.10 Supporting Links and Forms for Non-Hypermedia Services
4.11 Validating Data Properties At Runtime
4.12 Using Document Schemas to Validate Outgoing Messages
4.13 Using Document Queries to Validate Incoming Messages
4.14 Validating Incoming Data
4.15 Maintaining Your Own State
4.16 Having A Goal In Mind

Client Patterns



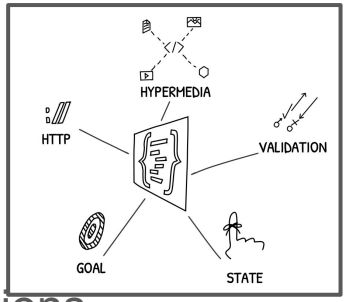
4.1 Limiting the use of Hard-Coded URLs
4.2 Code Clients to be HTTP-Aware
4.3 Coding More Resilient Clients With Message-Centric Implementations
4.4 Coding Effective Clients to Understand Vocabulary Profiles
4.5 Negotiate for Profile Support at Runtime
4.6 Managing Representation Formats At Runtime
4.7 Using Schema Documents as a Source of Message Metadata
4.8 Every Important Element Within a Response Needs an Identifier
4.9 Relying on Hypermedia Controls In the Response
4.10 Supporting Links and Forms for Non-Hypermedia Services
4.11 Validating Data Properties At Runtime
4.12 Using Document Schemas to Validate Outgoing Messages
4.13 Using Document Queries to Validate Incoming Messages
4.14 Validating Incoming Data
4.15 Maintaining Your Own State
4.16 Having A Goal In Mind

Client Patterns



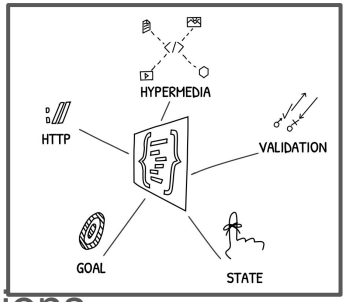
4.1 Limiting the use of Hard-Coded URLs
4.2 Code Clients to be HTTP-Aware
4.3 Coding More Resilient Clients With Message-Centric Implementations
4.4 Coding Effective Clients to Understand Vocabulary Profiles
4.5 Negotiate for Profile Support at Runtime
4.6 Managing Representation Formats At Runtime
4.7 Using Schema Documents as a Source of Message Metadata
4.8 Every Important Element Within a Response Needs an Identifier
4.9 Relying on Hypermedia Controls In the Response
4.10 Supporting Links and Forms for Non-Hypermedia Services
4.11 Validating Data Properties At Runtime
4.12 Using Document Schemas to Validate Outgoing Messages
4.13 Using Document Queries to Validate Incoming Messages
4.14 Validating Incoming Data
4.15 Maintaining Your Own State
4.16 Having A Goal In Mind

Client Patterns



4.1 Limiting the use of Hard-Coded URLs
4.2 Code Clients to be HTTP-Aware
4.3 Coding More Resilient Clients With Message-Centric Implementations
4.4 Coding Effective Clients to Understand Vocabulary Profiles
4.5 Negotiate for Profile Support at Runtime
4.6 Managing Representation Formats At Runtime
4.7 Using Schema Documents as a Source of Message Metadata
4.8 Every Important Element Within a Response Needs an Identifier
4.9 Relying on Hypermedia Controls In the Response
4.10 Supporting Links and Forms for Non-Hypermedia Services
4.11 Validating Data Properties At Runtime
4.12 Using Document Schemas to Validate Outgoing Messages
4.13 Using Document Queries to Validate Incoming Messages
4.14 Validating Incoming Data
4.15 Maintaining Your Own State
4.16 Having A Goal In Mind

Client Patterns



4.1 Limiting the use of Hard-Coded URLs
4.2 Code Clients to be HTTP-Aware
4.3 Coding More Resilient Clients With Message-Centric Implementations
4.4 Coding Effective Clients to Understand Vocabulary Profiles
4.5 Negotiate for Profile Support at Runtime
4.6 Managing Representation Formats At Runtime
4.7 Using Schema Documents as a Source of Message Metadata
4.8 Every Important Element Within a Response Needs an Identifier
4.9 Relying on Hypermedia Controls In the Response
4.10 Supporting Links and Forms for Non-Hypermedia Services
4.11 Validating Data Properties At Runtime
4.12 Using Document Schemas to Validate Outgoing Messages
4.13 Using Document Queries to Validate Incoming Messages
4.14 Validating Incoming Data
4.15 Maintaining Your Own State
4.16 Having A Goal In Mind

Client Patterns

Managing Representation Formats at Runtime

```
function handleResponse(ajax,url) {
  var ctype
  if(ajax.readyState===4) {
    try {
      ctype = ajax.getResponseHeader("content-type").toLowerCase();
      switch(ctype) {
        case "application/vnd.collection+json":
          cj.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.siren+json":
          siren.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.hal+json":
          hal.parse(JSON.parse(ajax.responseText));
          break;
        default:
          dump(ajax.responseText);
          break;
      }
    }
    catch(ex) {
      alert(ex);
    }
  }
}
```

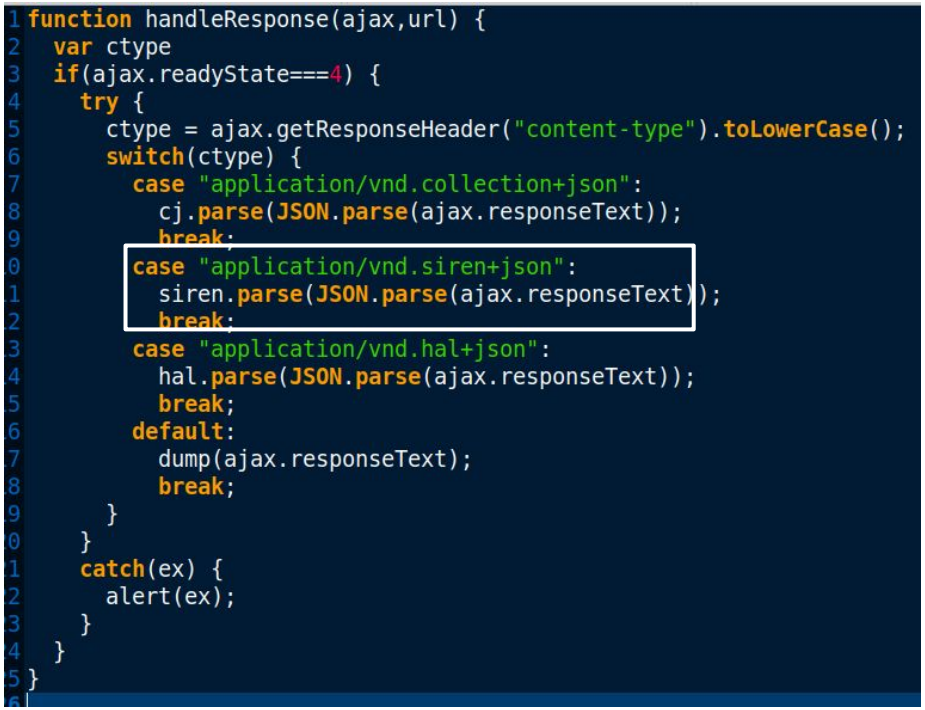
Client Patterns

Managing Representation Formats at Runtime

```
function handleResponse(ajax,url) {
  var ctype
  if(ajax.readyState===4) {
    try {
      ctype = ajax.getResponseHeader("content-type").toLowerCase();
      switch(ctype) {
        case "application/vnd.collection+json":
          cj.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.siren+json":
          siren.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.hal+json":
          hal.parse(JSON.parse(ajax.responseText));
          break;
        default:
          dump(ajax.responseText);
          break;
      }
    }
    catch(ex) {
      alert(ex);
    }
  }
}
```

Client Patterns

Managing Representation Formats at Runtime



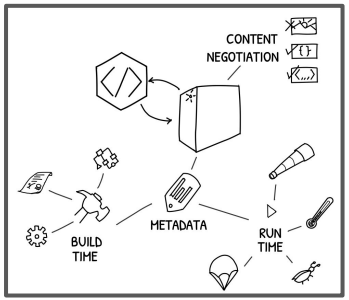
```
function handleResponse(ajax,url) {
  var ctype
  if(ajax.readyState===4) {
    try {
      ctype = ajax.getResponseHeader("content-type").toLowerCase();
      switch(ctype) {
        case "application/vnd.collection+json":
          cj.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.siren+json":
          siren.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.hal+json":
          hal.parse(JSON.parse(ajax.responseText));
          break;
        default:
          dump(ajax.responseText);
          break;
      }
    }
    catch(ex) {
      alert(ex);
    }
  }
}
```

Client Patterns

Managing Representation Formats at Runtime

```
function handleResponse(ajax,url) {
  var ctype
  if(ajax.readyState===4) {
    try {
      ctype = ajax.getResponseHeader("content-type").toLowerCase();
      switch(ctype) {
        case "application/vnd.collection+json":
          cj.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.siren+json":
          siren.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.hal+json":
          hal.parse(JSON.parse(ajax.responseText));
          break;
        default:
          dump(ajax.responseText);
          break;
      }
    }
    catch(ex) {
      alert(ex);
    }
  }
}
```

Make clients adaptable

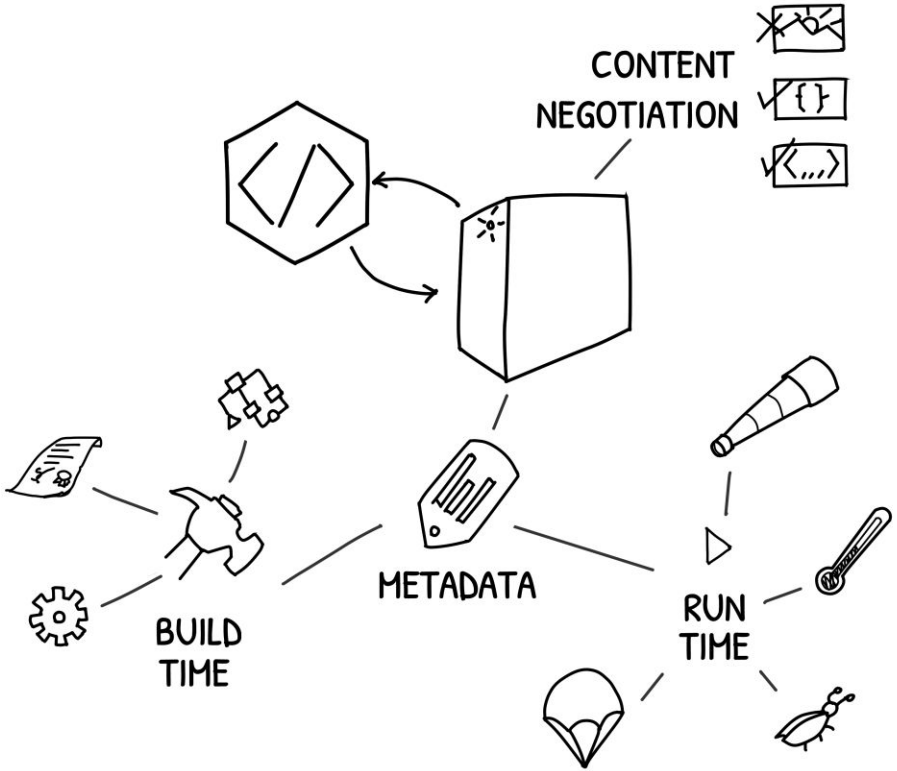


Services

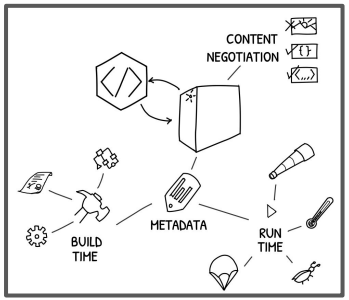
> The best software architecture "knows" what changes often and makes that easy.
>
> —Paul Clements

Service Patterns

The API is the contract — the promise that needs to be kept.

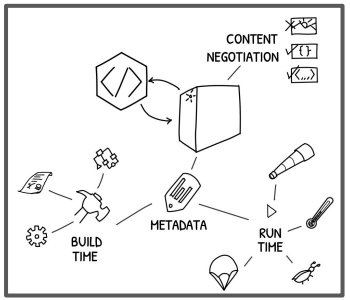


Service Patterns



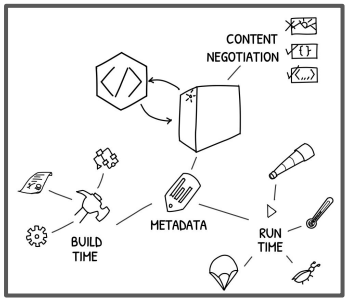
5.1 Publishing at Least One Stable URL
5.2 Preventing Internal Model Leaks
5.3 Converting Internal Models to External Messages
5.4 Expressing Internal Functions as External Actions
5.5 Advertising Support for Client Preferences for Responses
5.6 Supporting HTTP Content Negotiation
5.7 Publishing Complete Vocabularies for Machine Clients
5.8 Supporting Shared Vocabularies in Standard Formats
5.9 Publishing Service Definition Documents
5.10 Publishing API Metadata
5.11 Supporting Service Health Monitoring
5.12 Standardizing Error Reporting
5.13 Improve Service Discoverability with a Runtime Service Registry
5.14 Increasing Throughput with Client-Supplied Identifiers
5.15 Improving Reliability with Idempotent Create
5.16 Providing Runtime Fallbacks for Dependent Services
5.17 Using Semantic Proxies to Access Non-Compliant Services

Service Patterns



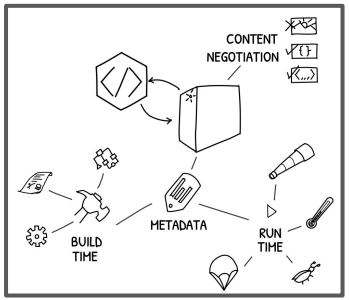
5.1 Publishing at Least One Stable URL
5.2 Preventing Internal Model Leaks
5.3 Converting Internal Models to External Messages
5.4 Expressing Internal Functions as External Actions
5.5 Advertising Support for Client Preferences for Responses
5.6 Supporting HTTP Content Negotiation
5.7 Publishing Complete Vocabularies for Machine Clients
5.8 Supporting Shared Vocabularies in Standard Formats
5.9 Publishing Service Definition Documents
5.10 Publishing API Metadata
5.11 Supporting Service Health Monitoring
5.12 Standardizing Error Reporting
5.13 Improve Service Discoverability with a Runtime Service Registry
5.14 Increasing Throughput with Client-Supplied Identifiers
5.15 Improving Reliability with Idempotent Create
5.16 Providing Runtime Fallbacks for Dependent Services
5.17 Using Semantic Proxies to Access Non-Compliant Services

Service Patterns



5.1 Publishing at Least One Stable URL
5.2 Preventing Internal Model Leaks
5.3 Converting Internal Models to External Messages
5.4 Expressing Internal Functions as External Actions
5.5 Advertising Support for Client Preferences for Responses
5.6 Supporting HTTP Content Negotiation
5.7 Publishing Complete Vocabularies for Machine Clients
5.8 Supporting Shared Vocabularies in Standard Formats
5.9 Publishing Service Definition Documents
5.10 Publishing API Metadata
5.11 Supporting Service Health Monitoring
5.12 Standardizing Error Reporting
5.13 Improve Service Discoverability with a Runtime Service Registry
5.14 Increasing Throughput with Client-Supplied Identifiers
5.15 Improving Reliability with Idempotent Create
5.16 Providing Runtime Fallbacks for Dependent Services
5.17 Using Semantic Proxies to Access Non-Compliant Services

Service Patterns



5.1 Publishing at Least One Stable URL
5.2 Preventing Internal Model Leaks
5.3 Converting Internal Models to External Messages
5.4 Expressing Internal Functions as External Actions
5.5 Advertising Support for Client Preferences for Responses
5.6 Supporting HTTP Content Negotiation
5.7 Publishing Complete Vocabularies for Machine Clients
5.8 Supporting Shared Vocabularies in Standard Formats
5.9 Publishing Service Definition Documents
5.10 Publishing API Metadata
5.11 Supporting Service Health Monitoring
5.12 Standardizing Error Reporting
5.13 Improve Service Discoverability with a Runtime Service Registry
5.14 Increasing Throughput with Client-Supplied Identifiers
5.15 Improving Reliability with Idempotent Create
5.16 Providing Runtime Fallbacks for Dependent Services
5.17 Using Semantic Proxies to Access Non-Compliant Services

Service Patterns

Improve Service Discoverability with a Runtime Service Registry

```
var srsResponse = null;
var srsRegister({Url:"...","name":"...", .....});

// register this service w/ defaults
discovery.register(srsRegister, function(data, response) {
  srsResponse = JSON.parse(data);
  initiateKeepAlive(srsResponse.href, srsResponse.milliseconds);
  http.createServer(uuidGenerator).listen(port);
  console.info('uuid-generator running on port '+port+'.');
});
```

Service Patterns

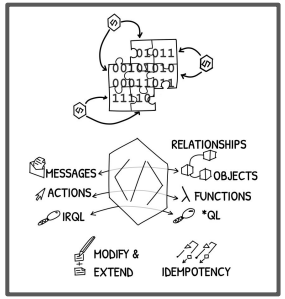
Improve Service Discoverability with a Runtime Service Registry

```
var srsResponse = null;
var srsRegister({Url:"...","name":"...", .....});

// register this service w/ defaults
discovery.register(srsRegister, funct
  srsResponse = JSON.parse(data);
  initiateKeepAlive(srsResponse.href,
  http.createServer(uuidGenerator).li
  console.info('uuid-generator runnin
});
```

```
// set up proper discovery shutdown
process.on('SIGTERM', function () {
  discovery.unregister(null, function(response) {
    try {
      uuidGenerator.close(function() {
      console.log('gracefully shutting down');
        process.exit(0);
      });
    } catch(e){}
  });
  setTimeout(function() {
    console.error('forcefully shutting down');
    process.exit(1);
  }, 10000);
});
```

Make services modifiable



Data

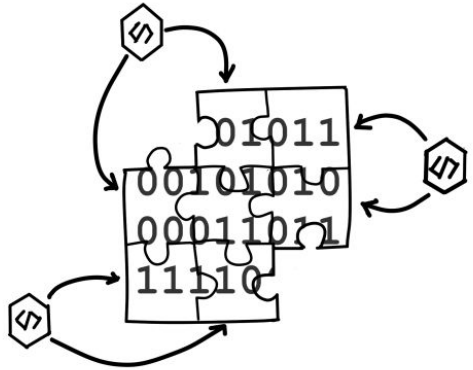
First step in breaking the data centric habit, is to stop designing systems as a collection of data services, and instead design for business capabilities.

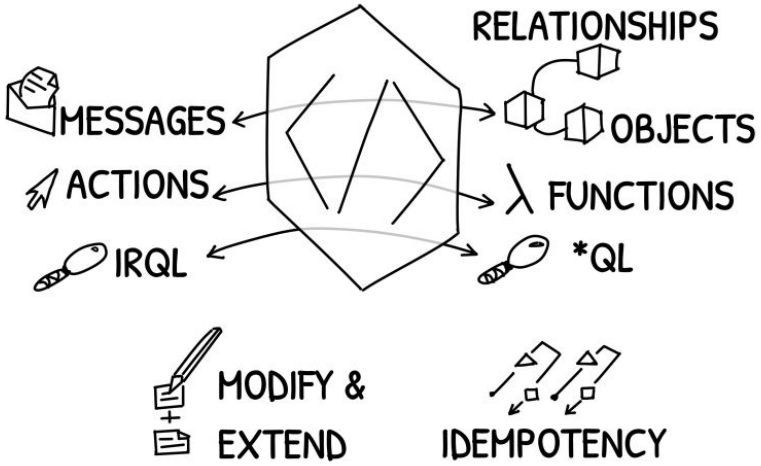
—Irakli Nadareishvili JPMorgan Chase

Data Patterns

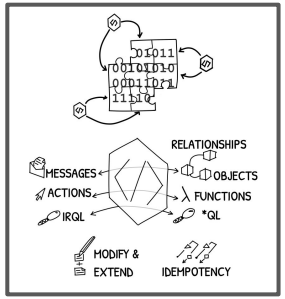
"Your data model is not your object model is not your resource model is not your representation model."

-- *Amundsen's Maxim*



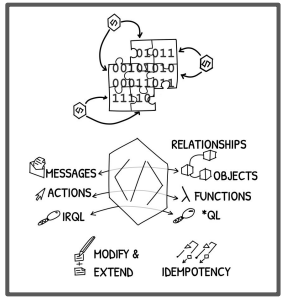


Data Patterns



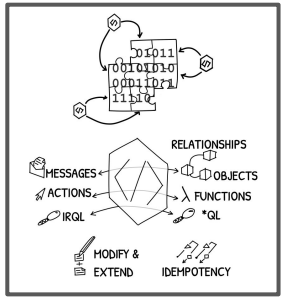
6.1 Hiding Your Data Storage Internals
6.2 Making All Changes Idempotent
6.3 Hide Data Relationships for External Actions
6.4 Leveraging HTTP URLs to Support “Contains” and “And” Queries
6.5 Returning Metadata for Query Responses
6.6 Returning HTTP 200 vs. HTTP 400 for Data-Centric Queries
6.7 Using Media Types for Data Queries
6.8 Ignore Unknown Data Fields
6.9 Improving Performance with Caching Directives
6.10 Modifying Data Models In Production
6.11 Extending Remote Data Stores
6.12 Limiting Large Scale Responses
6.13 Using Pass-Through Proxies for Data Exchange

Data Patterns



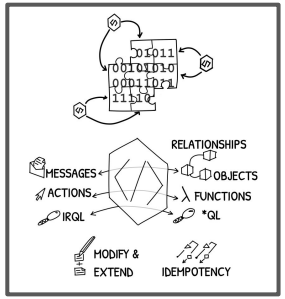
6.1 Hiding Your Data Storage Internals
6.2 Making All Changes Idempotent
6.3 Hide Data Relationships for External Actions
6.4 Leveraging HTTP URLs to Support “Contains” and “And” Queries
6.5 Returning Metadata for Query Responses
6.6 Returning HTTP 200 vs. HTTP 400 for Data-Centric Queries
6.7 Using Media Types for Data Queries
6.8 Ignore Unknown Data Fields
6.9 Improving Performance with Caching Directives
6.10 Modifying Data Models In Production
6.11 Extending Remote Data Stores
6.12 Limiting Large Scale Responses
6.13 Using Pass-Through Proxies for Data Exchange

Data Patterns



6.1 Hiding Your Data Storage Internals
6.2 Making All Changes Idempotent
6.3 Hide Data Relationships for External Actions
6.4 Leveraging HTTP URLs to Support “Contains” and “And” Queries
6.5 Returning Metadata for Query Responses
6.6 Returning HTTP 200 vs. HTTP 400 for Data-Centric Queries
6.7 Using Media Types for Data Queries
6.8 Ignore Unknown Data Fields
6.9 Improving Performance with Caching Directives
6.10 Modifying Data Models In Production
6.11 Extending Remote Data Stores
6.12 Limiting Large Scale Responses
6.13 Using Pass-Through Proxies for Data Exchange

Data Patterns

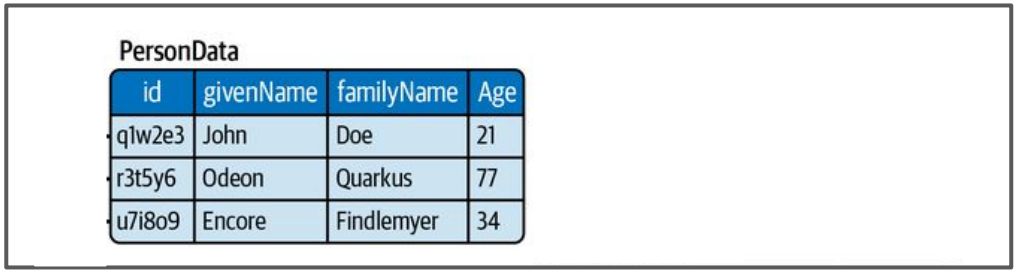


6.1 Hiding Your Data Storage Internals
6.2 Making All Changes Idempotent
6.3 Hide Data Relationships for External Actions
6.4 Leveraging HTTP URLs to Support “Contains” and “And” Queries
6.5 Returning Metadata for Query Responses
6.6 Returning HTTP 200 vs. HTTP 400 for Data-Centric Queries
6.7 Using Media Types for Data Queries
6.8 Ignore Unknown Data Fields
6.9 Improving Performance with Caching Directives
6.10 Modifying Data Models In Production
6.11 Extending Remote Data Stores
6.12 Limiting Large Scale Responses
6.13 Using Pass-Through Proxies for Data Exchange

Data Patterns

Modifying Data Models in Production

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21
}
```



Data Patterns

Modifying Data Models in Production

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21
}
```

```
{
  "givenName": "John",
  "middleName": "Seymore",
  "familyName": "Doe",
  "age": 21
};
```

PersonData

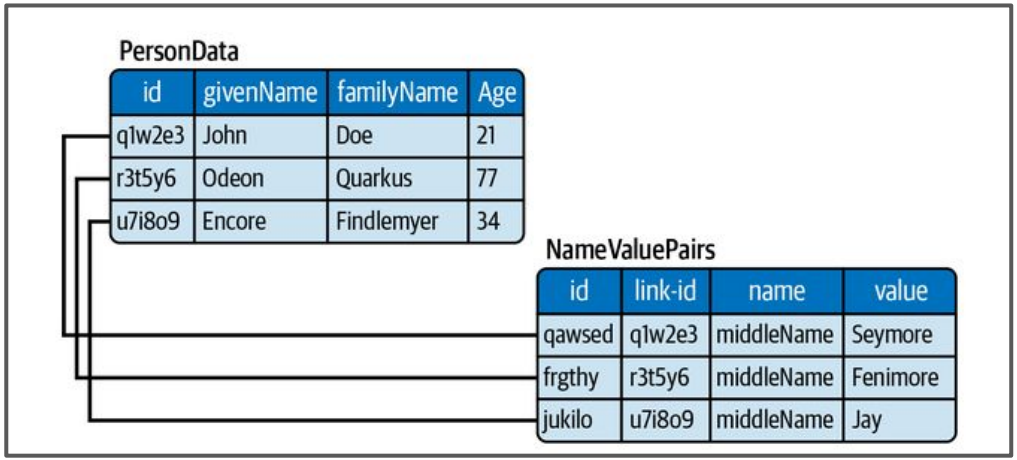
id	givenName	familyName	Age
q1w2e3	John	Doe	21
r3t5y6	Odeon	Quarkus	77
u7i8o9	Encore	Findlemyer	34

Data Patterns

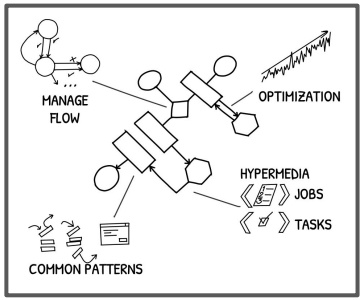
Modifying Data Models in Production

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21
}
```

```
{
  "givenName": "John",
  "middleName": "Seymore",
  "familyName": "Doe",
  "age": 21
};
```



Make data portable

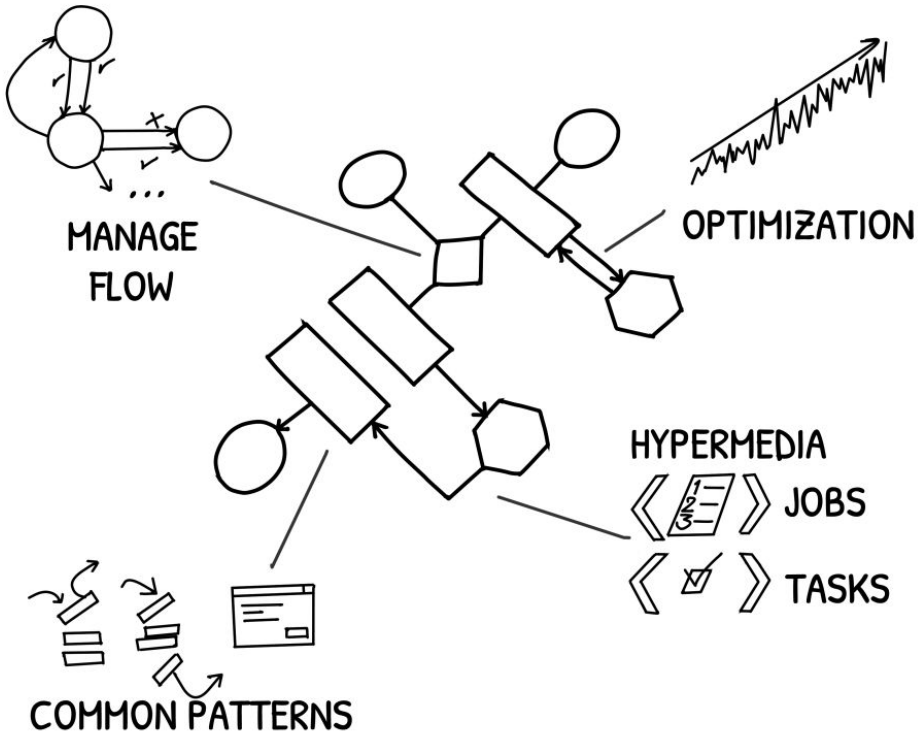


Workflow

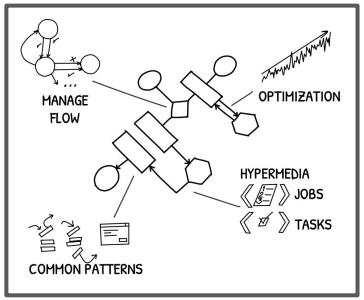
> Productivity is never an accident. It is always the result of a commitment to excellence, intelligent planning, and focused effort.
>
> —Paul J. Meyer

Workflow Patterns

Each service that is enlisted in a workflow should be a ***composable*** *service.*

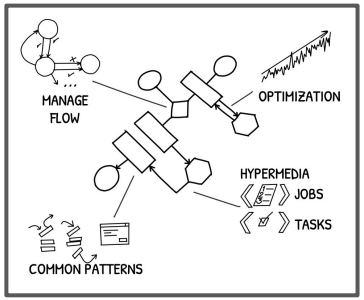


Workflow Patterns



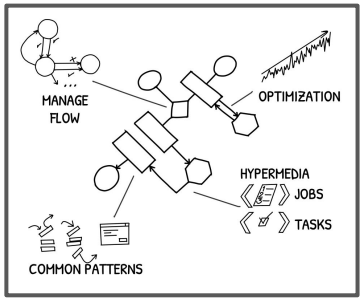
7.1 Designing Workflow-Compliant Services
7.2 Supporting Shared State for Workflows
7.3 Describing Workflow as Code
7.4 Describing Workflow as DSL
7.5 Describing Workflow as Documents
7.6 Supporting RESTful Job Control Language
7.7 Exposing a Progress Resource for Your Workflows
7.8 Returning All Related Actions
7.9 Returning Most-Recently Used Resources (MRUs)
7.10 Supporting Stateful Work-In-Progress
7.11 Enabling Standard List Navigation
7.12 Supporting Partial Form Submit
7.13 Using State-Watch to Enable Client-Driven Workflow
7.14 Optimizing Queries With Stored Replays
7.15 Synchronous Reply for Incomplete Work with 202 Accepted
7.16 Short-Term Fixes with Automatic Retries
7.17 Supporting Local Undo/Rollback
7.18 Calling for Help
7.19 Scaling Workflow with Queues and Clusters
7.20 Using Workflow Proxies to Enlist Non-Compliant Services

Workflow Patterns



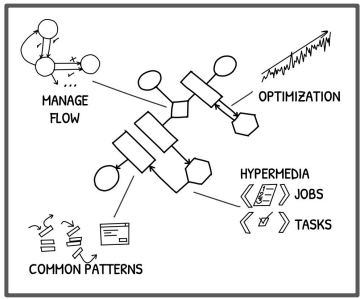
7.1 Designing Workflow-Compliant Services
7.2 Supporting Shared State for Workflows
7.3 Describing Workflow as Code
7.4 Describing Workflow as DSL
7.5 Describing Workflow as Documents
7.6 Supporting RESTful Job Control Language
7.7 Exposing a Progress Resource for Your Workflows
7.8 Returning All Related Actions
7.9 Returning Most-Recently Used Resources (MRUs)
7.10 Supporting Stateful Work-In-Progress
7.11 Enabling Standard List Navigation
7.12 Supporting Partial Form Submit
7.13 Using State-Watch to Enable Client-Driven Workflow
7.14 Optimizing Queries With Stored Replays
7.15 Synchronous Reply for Incomplete Work with 202 Accepted
7.16 Short-Term Fixes with Automatic Retries
7.17 Supporting Local Undo/Rollback
7.18 Calling for Help
7.19 Scaling Workflow with Queues and Clusters
7.20 Using Workflow Proxies to Enlist Non-Compliant Services

Workflow Patterns



7.1 Designing Workflow-Compliant Services
7.2 Supporting Shared State for Workflows
7.3 Describing Workflow as Code
7.4 Describing Workflow as DSL
7.5 Describing Workflow as Documents
7.6 Supporting RESTful Job Control Language
7.7 Exposing a Progress Resource for Your Workflows
7.8 Returning All Related Actions
7.9 Returning Most-Recently Used Resources (MRUs)
7.10 Supporting Stateful Work-In-Progress
7.11 Enabling Standard List Navigation
7.12 Supporting Partial Form Submit
7.13 Using State-Watch to Enable Client-Driven Workflow
7.14 Optimizing Queries With Stored Replays
7.15 Synchronous Reply for Incomplete Work with 202 Accepted
7.16 Short-Term Fixes with Automatic Retries
7.17 Supporting Local Undo/Rollback
7.18 Calling for Help
7.19 Scaling Workflow with Queues and Clusters
7.20 Using Workflow Proxies to Enlist Non-Compliant Services

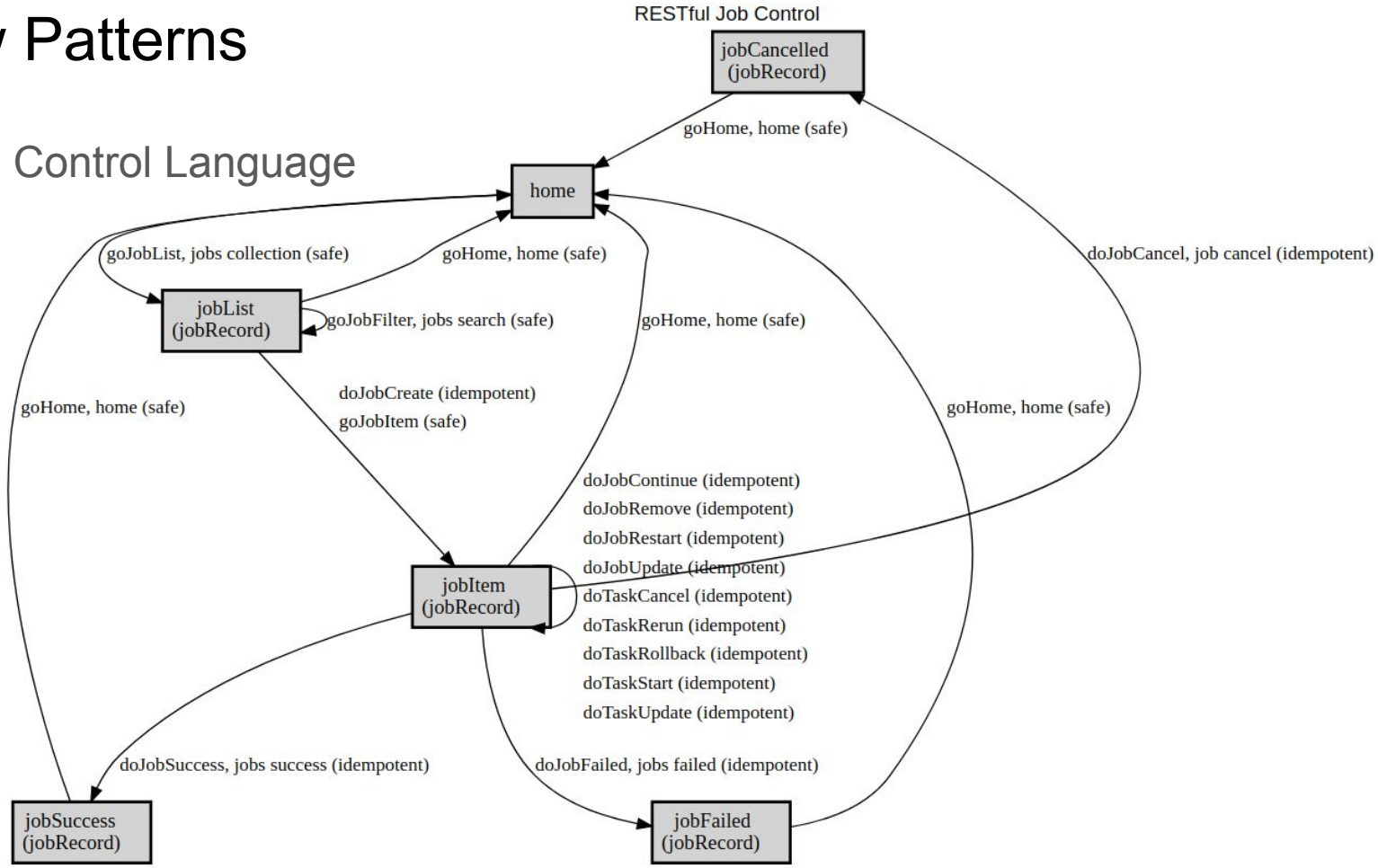
Workflow Patterns



7.1 Designing Workflow-Compliant Services
7.2 Supporting Shared State for Workflows
7.3 Describing Workflow as Code
7.4 Describing Workflow as DSL
7.5 Describing Workflow as Documents
7.6 Supporting RESTful Job Control Language
7.7 Exposing a Progress Resource for Your Workflows
7.8 Returning All Related Actions
7.9 Returning Most-Recently Used Resources (MRUs)
7.10 Supporting Stateful Work-In-Progress
7.11 Enabling Standard List Navigation
7.12 Supporting Partial Form Submit
7.13 Using State-Watch to Enable Client-Driven Workflow
7.14 Optimizing Queries With Stored Replays
7.15 Synchronous Reply for Incomplete Work with 202 Accepted
7.16 Short-Term Fixes with Automatic Retries
7.17 Supporting Local Undo/Rollback
7.18 Calling for Help
7.19 Scaling Workflow with Queues and Clusters
7.20 Using Workflow Proxies to Enlist Non-Compliant Services

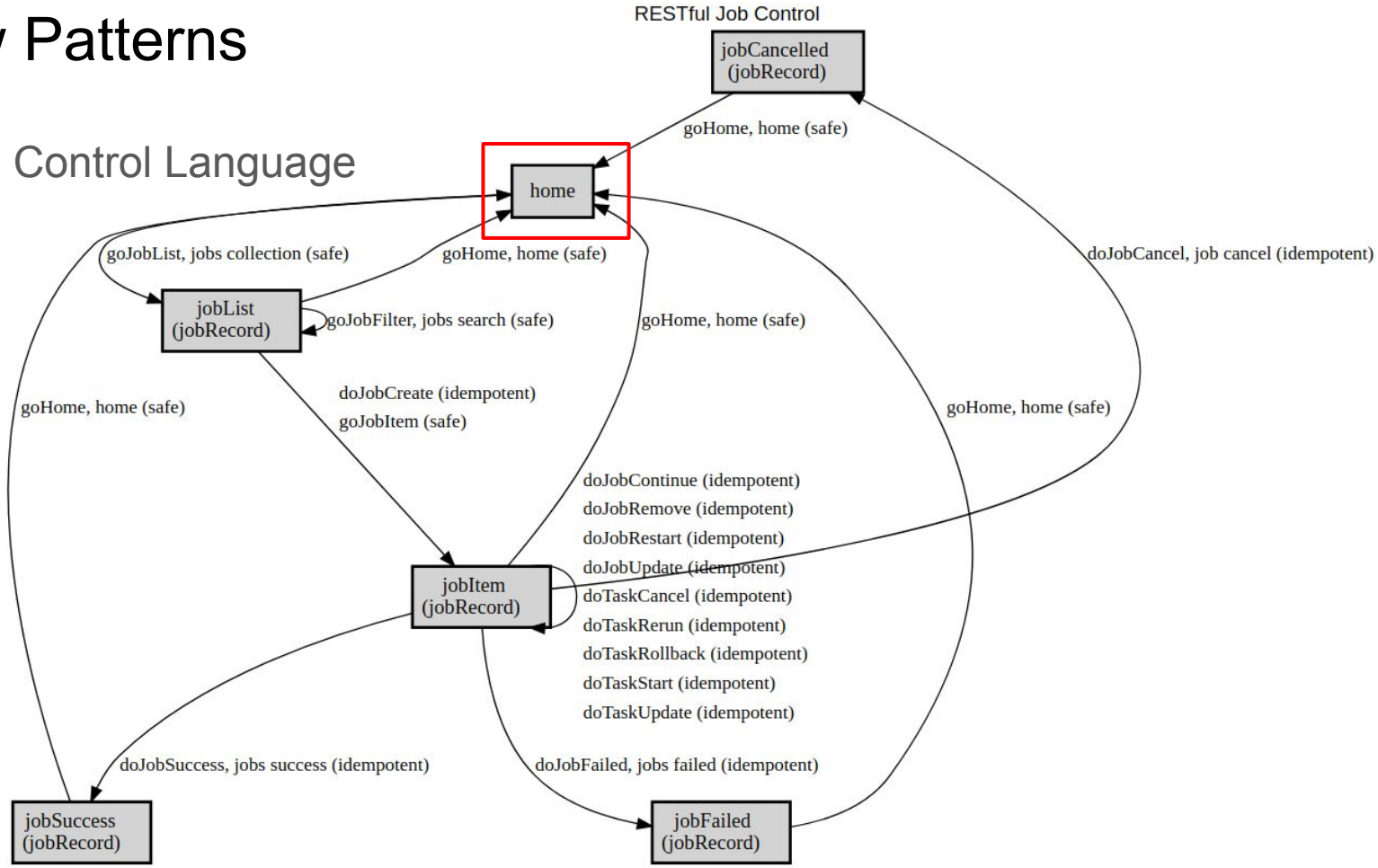
Workflow Patterns

RESTful Job Control Language



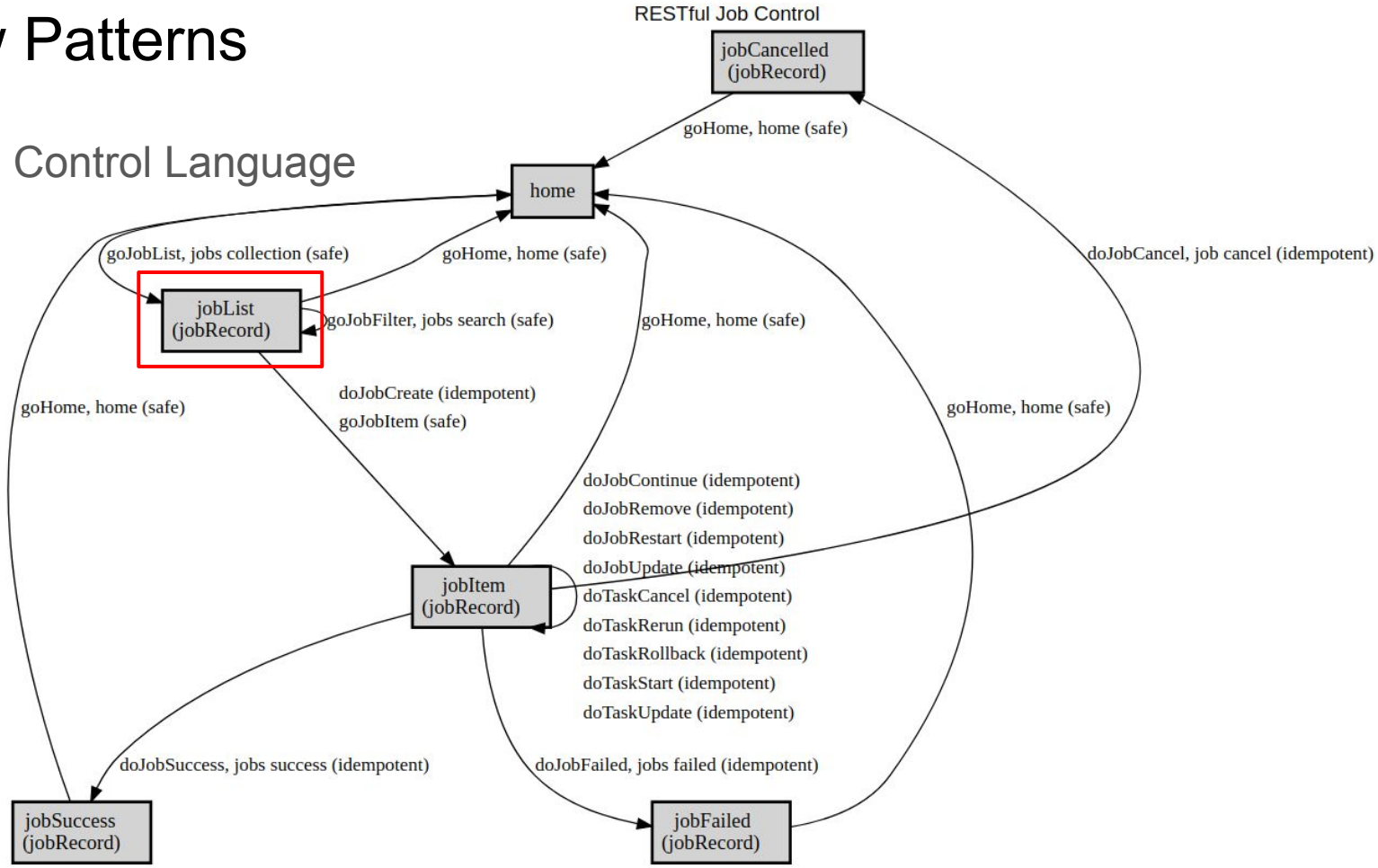
Workflow Patterns

RESTful Job Control Language



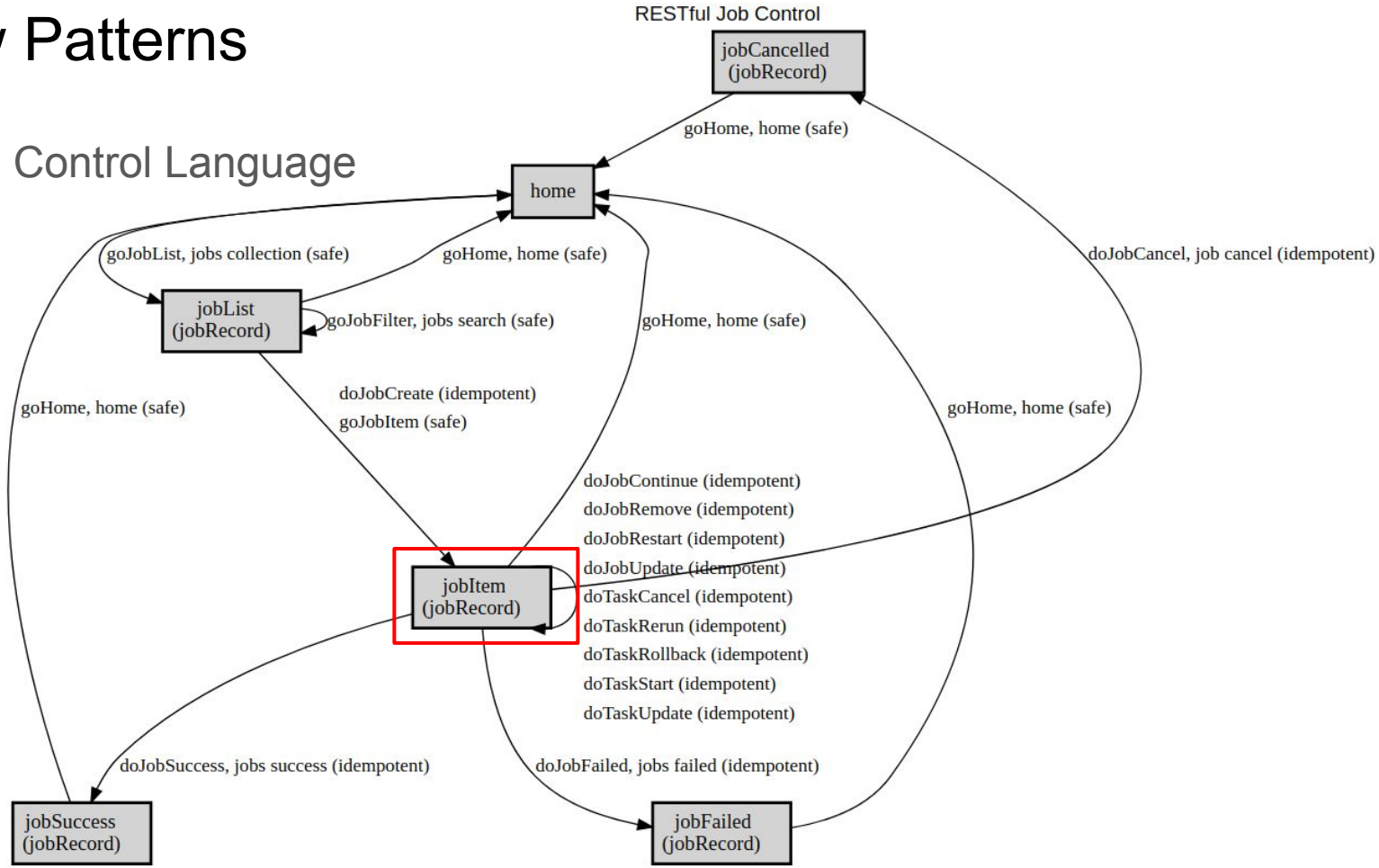
Workflow Patterns

RESTful Job Control Language



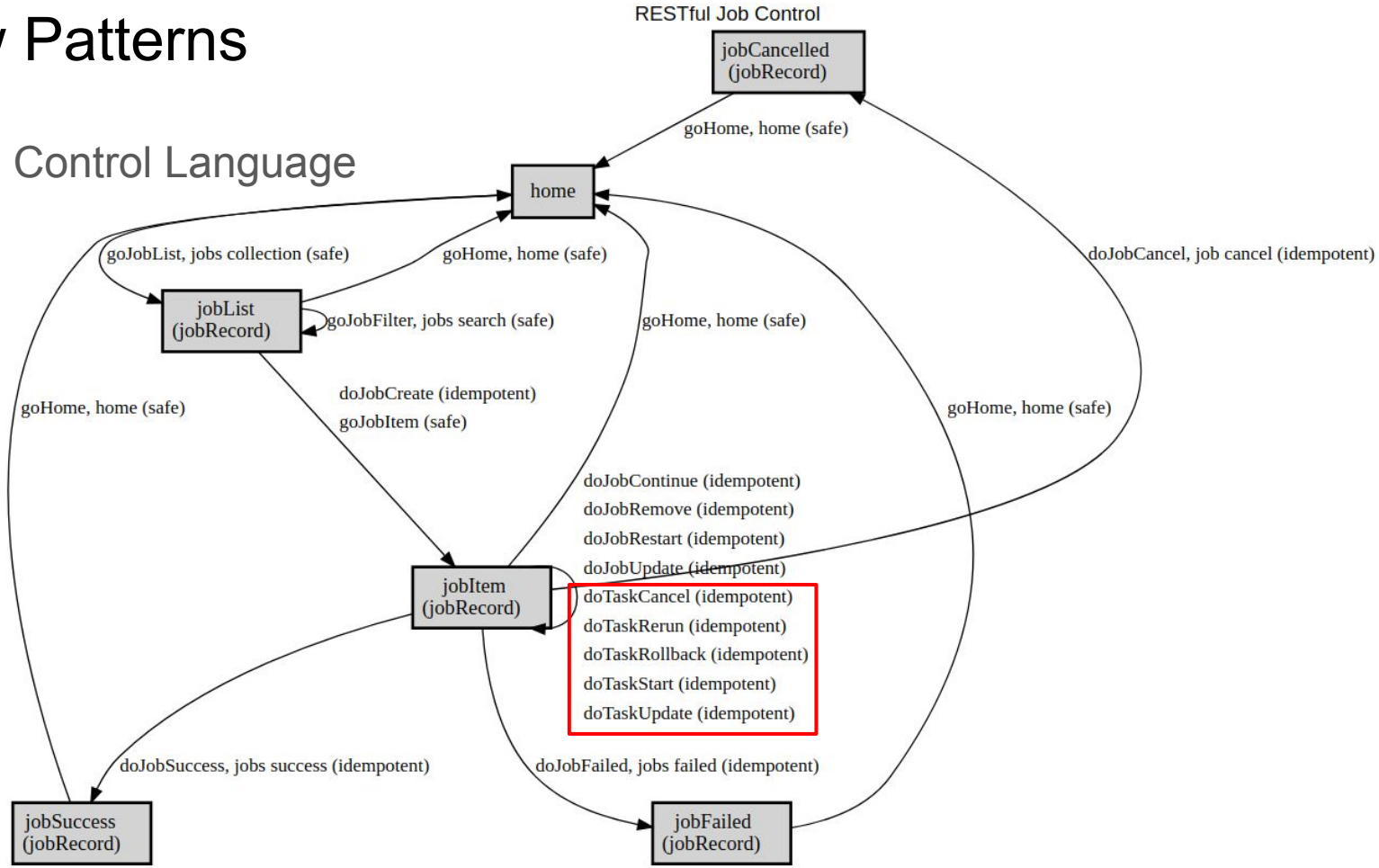
Workflow Patterns

RESTful Job Control Language



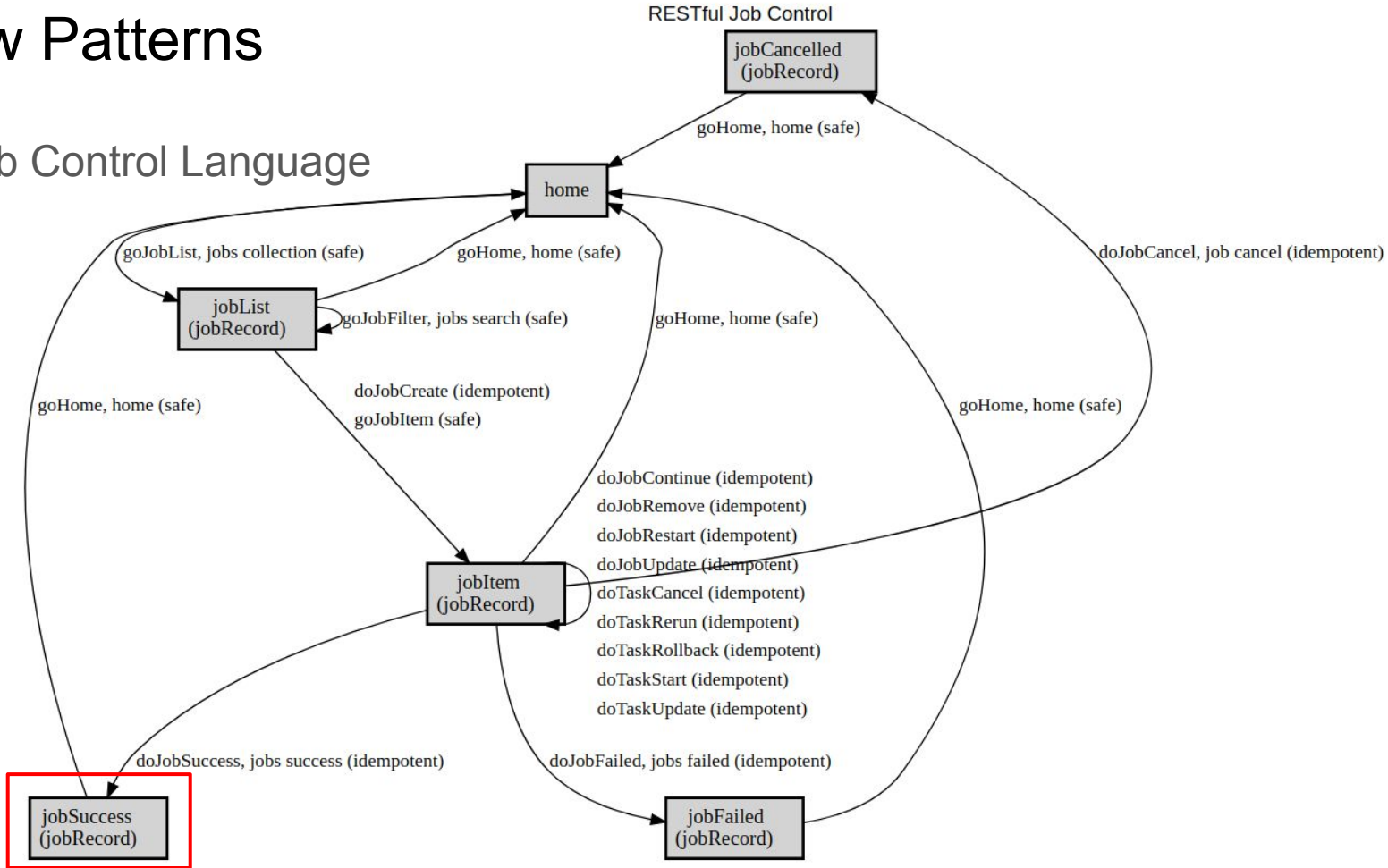
Workflow Patterns

RESTful Job Control Language



Workflow Patterns

RESTful Job Control Language



Make workflow flexible

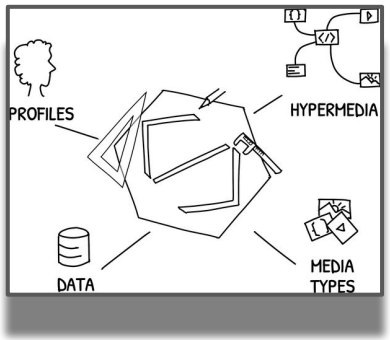
And so …

Goals

- Make designs composable
- Make clients adaptable
- Make services modifiable
- Make data portable
- Make workflow flexible

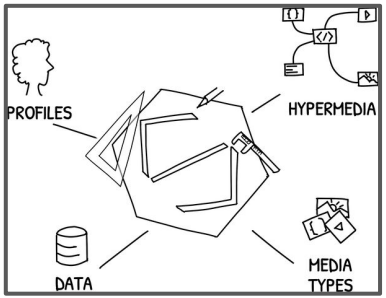
Goals

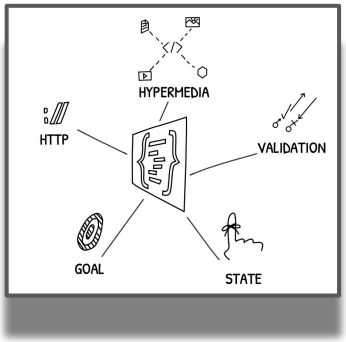
- **Make designs composable**
- Make clients adaptable
- Make services modifiable
- Make data portable
- Make workflow flexible



Goals

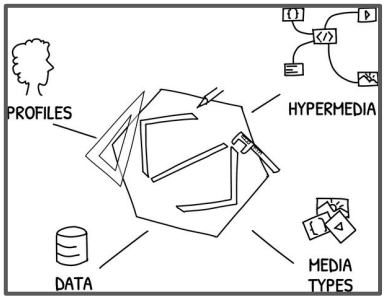
- Make designs composable
- **Make clients adaptable**
- Make services modifiable
- Make data portable
- Make workflow flexible

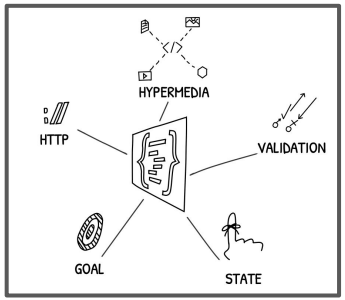


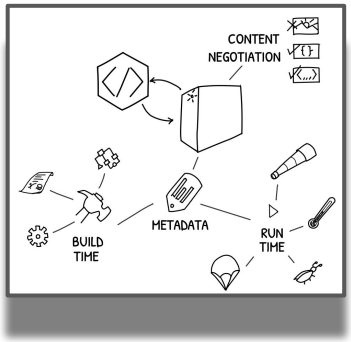


Goals

- Make designs composable
- Make clients adaptable
- **Make services modifiable**
- Make data portable
- Make workflow flexible

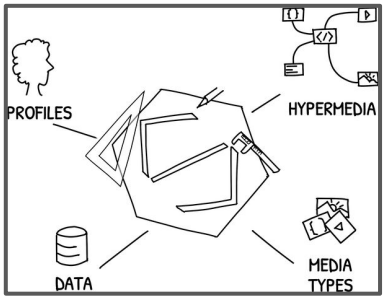


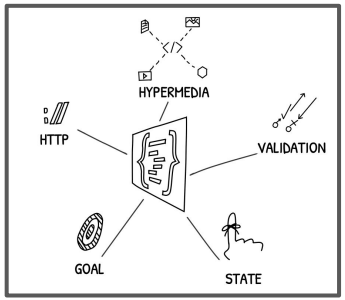


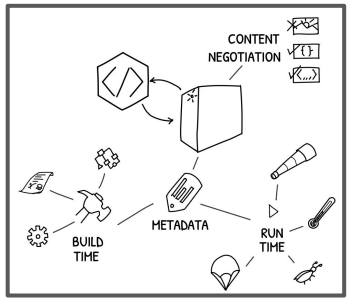


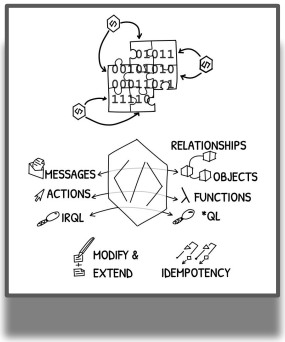
Goals

- Make designs composable
- Make clients adaptable
- Make services modifiable
- **Make data portable**
- Make workflow flexible



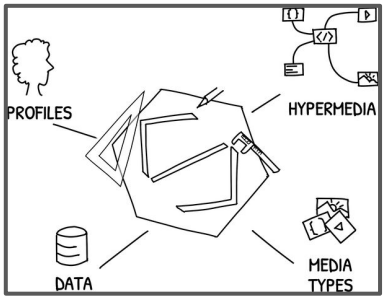


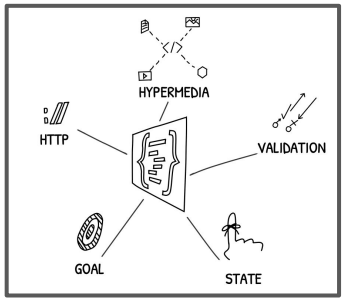


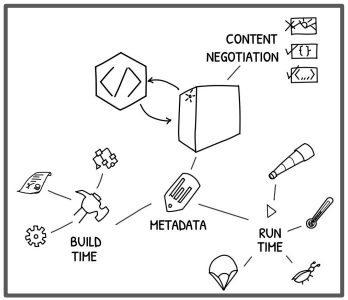


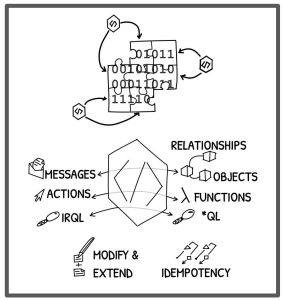
Goals

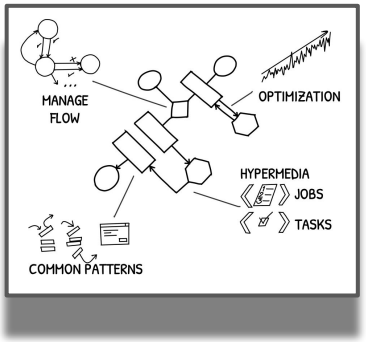
- Make designs composable
- Make clients adaptable
- Make services modifiable
- Make data portable
- **Make workflow flexible**





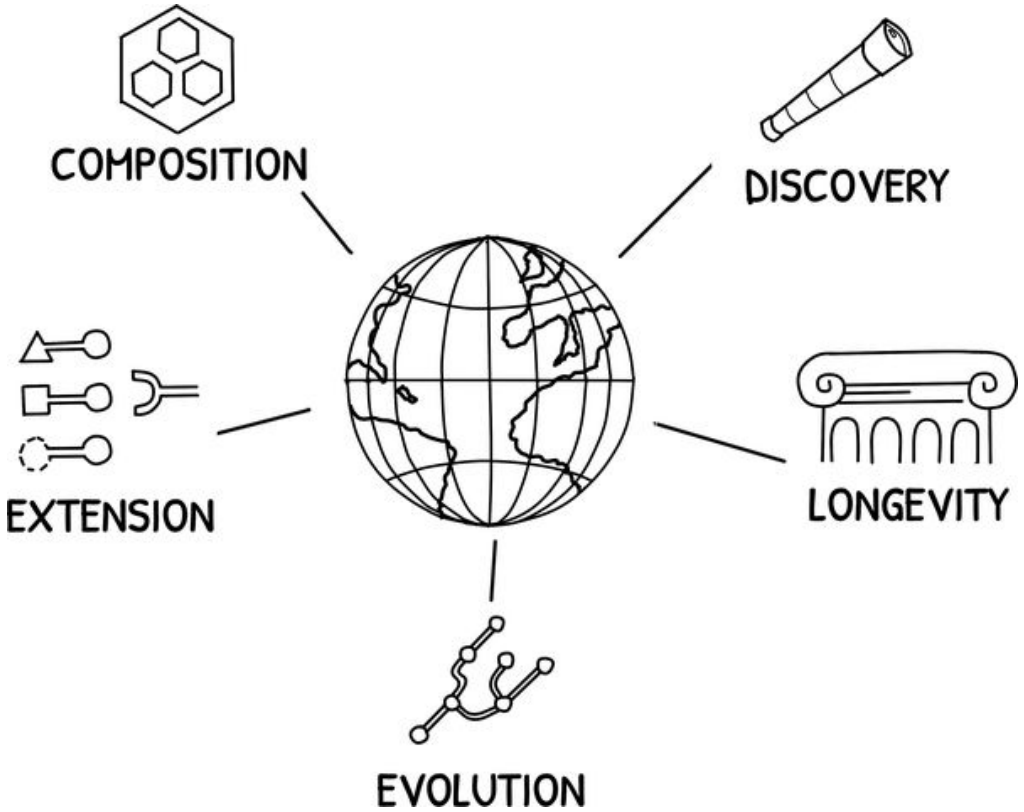






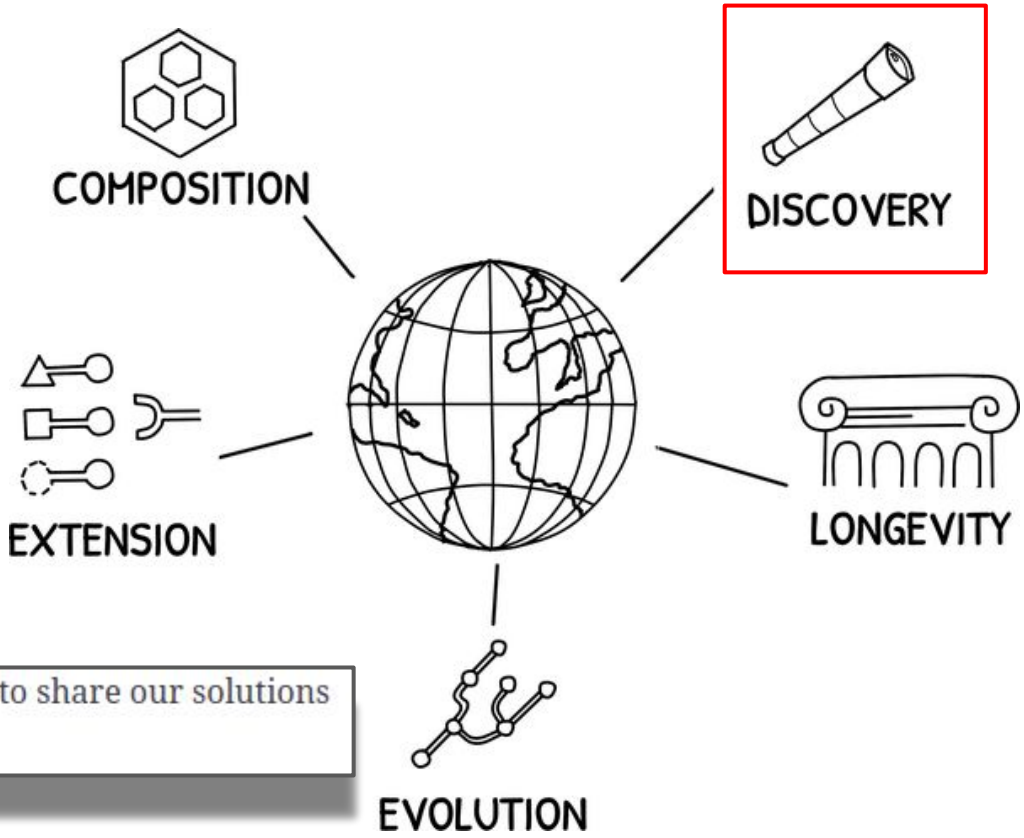
The RESTful Web API Principle

"Leverage global reach to solve problems you haven't thought of for people you have never met."



The RESTful Web API Principle

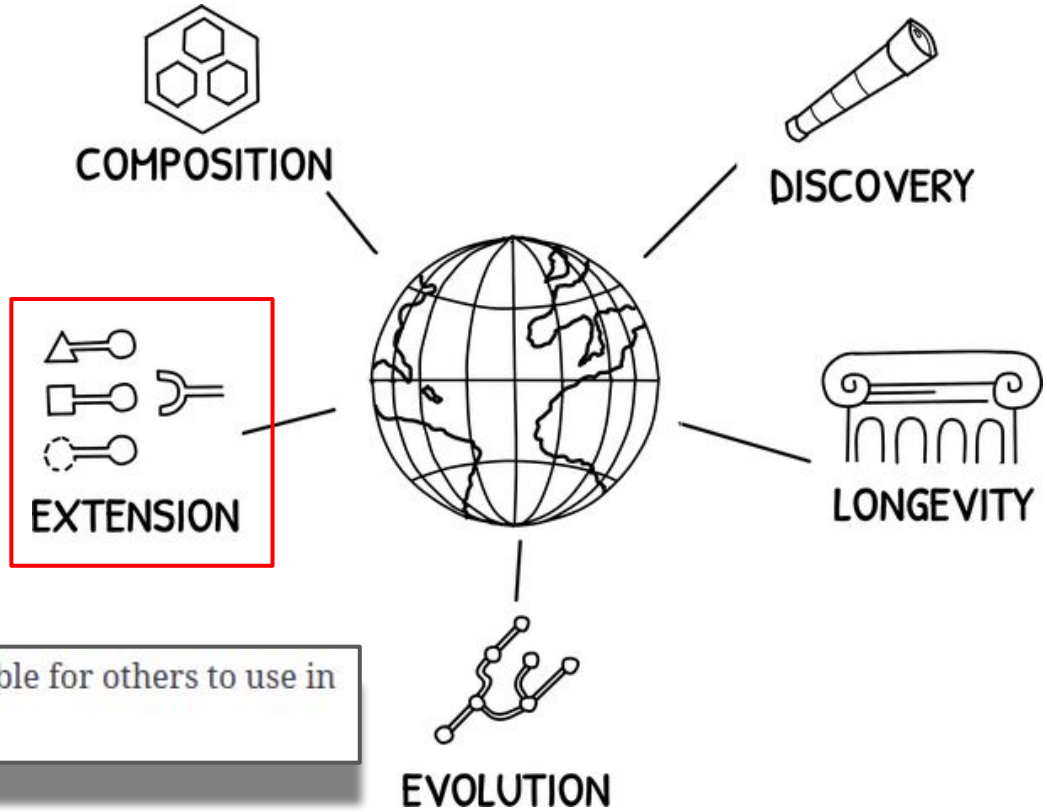
"Leverage global reach to solve problems you haven't thought of for people you have never met."



> Good recipes increase our global reach—the ability to share our solutions and to find and use the solutions of others.

The RESTful Web API Principle

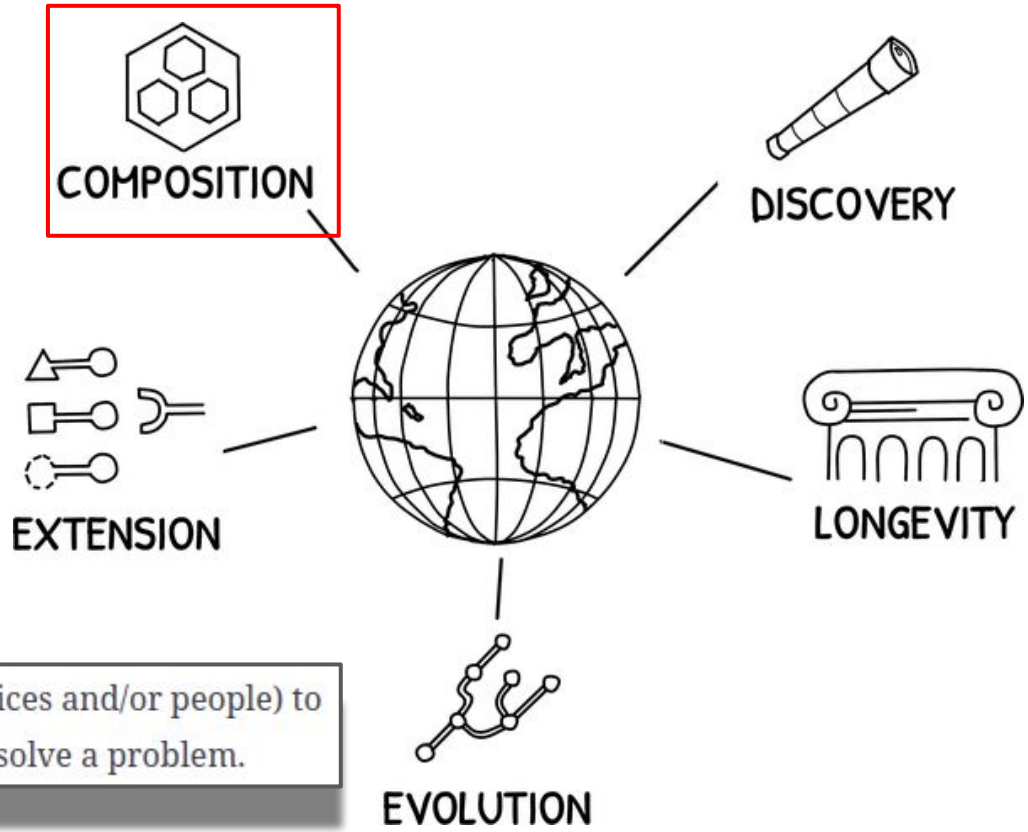
"Leverage global reach to solve problems you haven't thought of for people you have never met."



> Good recipes make well-designed services available for others to use in ways we haven't thought of yet.

The RESTful Web API Principle

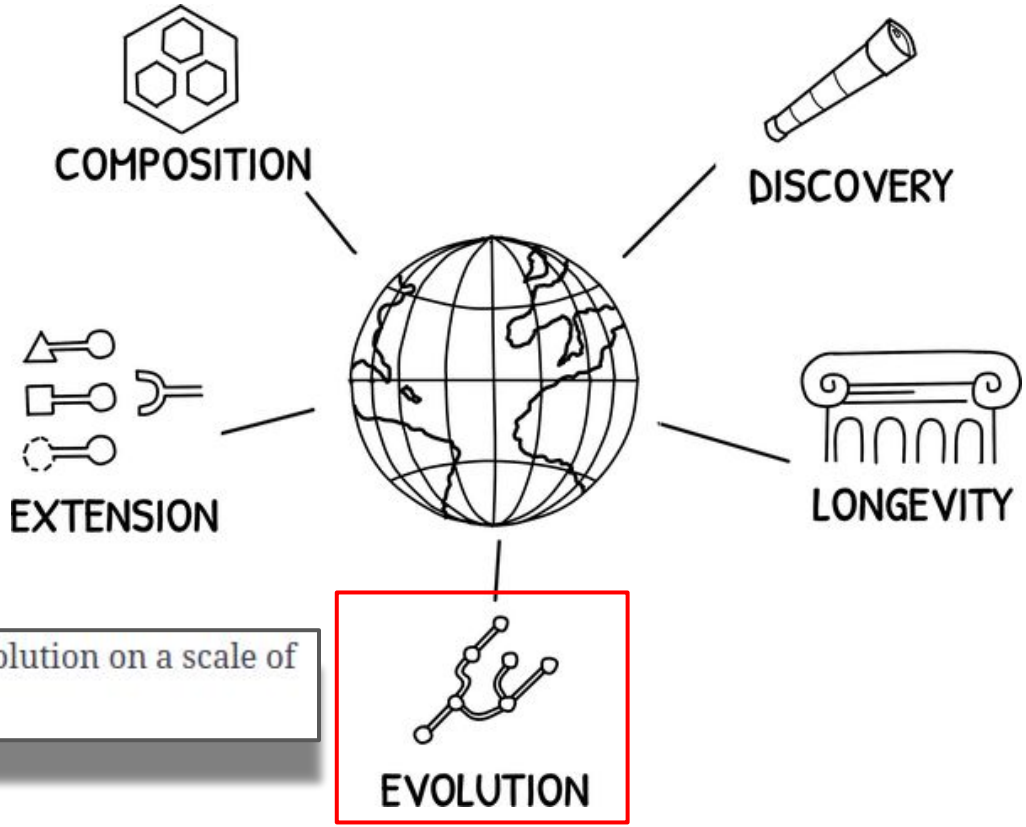
"Leverage global reach to solve problems you haven't thought of for people you have never met."



Good recipes make it possible for "strangers" (services and/or people) to safely and successfully interact with each other to solve a problem.

The RESTful Web API Principle

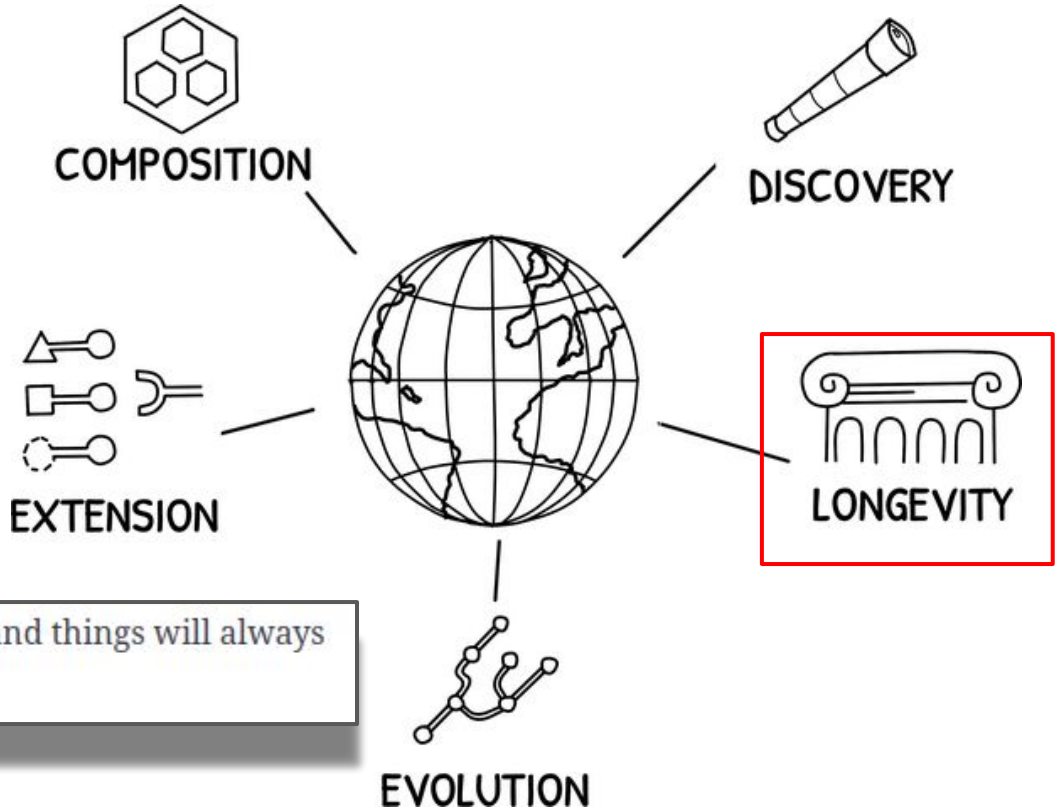
"Leverage global reach to solve problems you haven't thought of for people you have never met."



Good recipes promote longevity and independent evolution on a scale of decades.

The RESTful Web API Principle

"Leverage global reach to solve problems you haven't thought of for people you have never met."



Good recipes recognize that nothing is permanent and things will always change over time.

Pattern Thinking -- and Models

"Everything we think we know about the world is a model."

-- Donella Meadows, 2008

Pattern Thinking

"The difference between the novice and the teacher is simply that the novice has not learnt, yet, how to do things in such a way that they can afford to make small mistakes."

-- Christopher Alexander

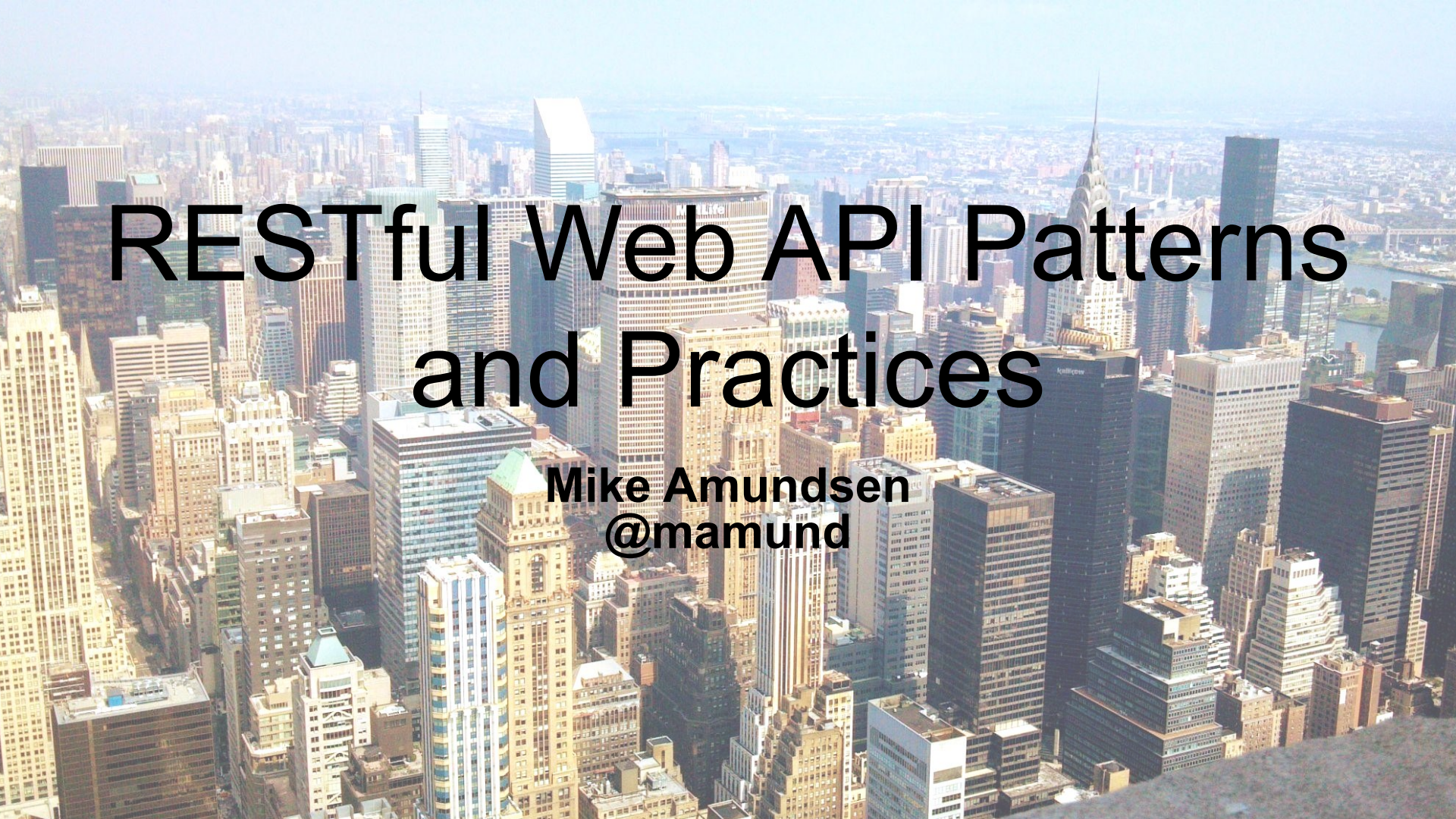
Pattern Thinking

*"The difference between the **novice** and the **teacher** is simply that the novice has not learnt, yet, how to do things in such a way that they can afford to make **small mistakes**."*

-- Christopher Alexander

RESTful Web API Patterns and Practices

Mike Amundsen
@mamund



RESTful Web API Patterns and Practices

Mike Amundsen
@mamund