REST API

The REST architectural style has been chosen for the API platform. At the same time, many business processes are asynchronous, which makes it impossible to fully implement them on the basis of HTTP as a synchronous protocol. Therefore, the HTTP API is extended with events in Kafka to notify about changes in resource state.

Key principles

It is very important to follow consistent principles for REST API:

1. All changes in the system are represented by standard CRUD operations on resources.

2. Each resource is uniquely identifiable (it has a unique ID). For example, in the Marqeta API, the resource identifier is called token.

3. For CRUD operations, the full capabilities of the HTTP protocol are used (methods, status codes, content type management, etc.).

4. The server implementation remains stateless (no client state is stored, only resource state).

5. The state of resources can be cached on intermediate hosts if the resource has been marked as cacheable by HTTP metadata.

6. The request and response must contain complete information for their processing (each request is considered independent).

7. Idempotency of all operations on resources (clients can make the same call repeatedly with the same result on the server).

Design rules

1. Use nouns but not verbs. HTTP methods are used to represent operations.

2. GET method and query parameters must not alter the state of resources.

3. Use HTTP headers to specify data serialization format. *Content-Type* and *Accept* headers define request/response formats.

4. Provide filtering, sorting, field selection, and paging for collections:

API examples with filtering and sorting

```
GET /cars?color=red Returns a list of red cars
GET /cars?seats<=2 Returns a list of cars with a maximum of 2 seats
GET /cars?sort=-manufactorer,+model
GET /cars?fields=manufacturer,model,id,color
GET /cars?offset=10&limit=5
```

5. Version API according to the common versioning approach.

6. Handle Errors with HTTP status codes.

7. Use the error payload to specify the root of the error.

Error payload example

```
{
  "errors": [
   {
    "userMessage": "Sorry, the requested resource does not exist",
    "internalMessage": "No car found in the database",
    "code": 34,
    "more info": "http://dev.mwaysolutions.com/blog/api/v1/errors/12345"
   }
  ]
}
```

Kafka events usage to extend REST API

For asynchronous scenarios, event streams are used in addition to the HTTP protocol. Key principles for using this approach:

1. Each resource type corresponds to a single channel of events streaming.

2. All events for a resource type have the same structure and carry information about changes in the state of a particular resource.

3. Each event has its own identifier to provide tracking and implement client-side idempotent processing.

4. Each event is linked to a specific resource through its identifier.

5. All events for a specific resource are strictly time-ordered and time-stamped.

Naming rules

All resources are named according to the entities of the domain model in order to best match business processes, as well as distribute the implementation on the server between microservices. Examples: *Card*, *Card Transition*, *Limit Request*.

HTTP endpoints

- For each resource type, a root endpoint is created to create and retrieve a list of resources of this type. Its name is the same as the plural name of the resource type.

  For example, for resource type *Card* the root endpoint will be */cards*, allowing to get a list of resources with this type via HTTP GET and create new resources via HTTP POST.

- To access a specific resource, a child endpoint with a resource identifier is used.

  For example, for resource type *Card* the child will be */cards/${card_id}*, allowing to get a concrete resource state via HTTP GET and update its state via HTTP PUT.

- To provide operations with only a part of a specific resource, deeper child endpoints can be created for named part of the resource.

  For example, to manage PIN for a resource with type *Card* child endpoint */cards/${card_id}/pin* will be created, allowing to get partial state of the resource via HTTP GET and perform partial state update via HTTP PUT.

Kafka topics

- Kafka topics are used to organize event streams for all types of resources. A separate Kafka topic is used for each type of resource. Topics are named by the name of the resource type.

  For example, for resource type *Card* topic will have the name *card-events*.

- Suffixes in topic names will also be used for different environments.

  For example, for resource type *Card* topic *card-events* in the DEV environment will have name *card-events-dev*, and in UAT environment *card-events-uat*.

- Each event in the topic has its own identifier, resource identifier, creation time, and changes in the resource state.

  For example, *event_id* for event unique identifier, *event_creation_date* for time-stamping, *token* or *id* for resource identifier, *data* or *state* for resource state.

- All events for the same resource must use the same Kafka partition key to allow sequential events processing on the consumer side. A resource identifier could be used easily for this purpose.

Spring Web MVC usage

1. All REST controllers must use dedicated DTOs for inputs and outputs. Domain and persistent entities are not allowed to avoid strong coupling.

2. All inputs are validated according to the API contract.

3. REST controllers don't have any business logic, they are responsible only for HTTP adaptor implementation and business service invocation with correct data.

4. If the API-first approach is used all REST controllers must be inherited from the interfaces generated from OpenAPI specifications. DTOs in this case are also taken from the generated JARs.

5. Common error handlers must be implemented in a separate class annotated as ControllerAdvice. Specific error handlers could be placed inside the particular controller class.

6. Each REST controller must restrict content types for inputs and outputs to supported ones (JSON by default) to avoid serialization issues.