Service API Frameworks

An overview of REST API Frameworks,
their design considerations and best practices.

Content

1. Reminder: REST-API
2. REST-API Framework Comparison
3. Routing
4. Middleware (API)
5. Validation
6. Authentication
7. Concept of Integrating External / Describing REST-API
8. Typical Project Structure
9. Recommended Design Patterns
10. Security Measures and best Practices
11. Sources

Reminder: REST-API

- Handles requests from external consumers
- Stateless:
  Data is sent and evaluated between systems independent of their state
- Standard response format in JSON or XML
- Allows access with resources via URI
- HTTP-verbs:
  GET, POST, PUT, DELETE

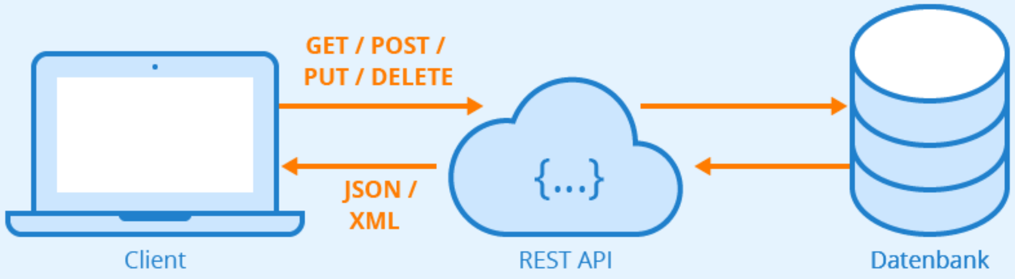


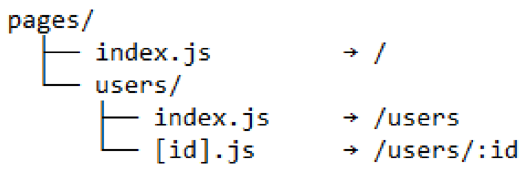
Figure: REST-API - Author: Seobility - License: Creative Commons Lizenz BY-SA 4.0

REST-API Framework Comparison

	Express.js	Spring Boot	FastAPI	Next.js	Quarkus
Language	JavaScript	Java / Kotlin	Python	JavaScript	Java / Kotlin
Performance	Excels in real-time environments	Can handle a large quantity of requests	Can handle a large quantity of requests with low latency	Excels in frontend performance and server optimizations	Fast Startup and low resource consumption
Ease of Use	Easy syntax and substantial middleware support	Directly linked to Java's complexity	Provides auto documentation and type hinting	Provides native server-side rendering	Requires minimal configuration
Use Case	Web-apps and APIs	Web-apps and microservices	APIs	Full-stack	Cloud-native Java microservices
Community	Large; Node.js ecosystem	Extensive; Java community	Growing; Python community	Large; React/Vercel ecosystem	Growing; Kubernetes-native focus

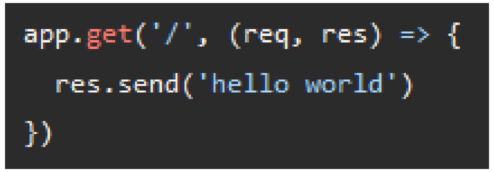
Routing

REST routes allow external consumers to call a specific method within an API. Those calls are made with one of the HTTP-verbs in combination with an URI. Usually a method is marked for the REST API with the wished HTTP-mapping with the URI path for that resource.
Here, Next.js is an exception, where the exact routing corresponds to the filesystem .



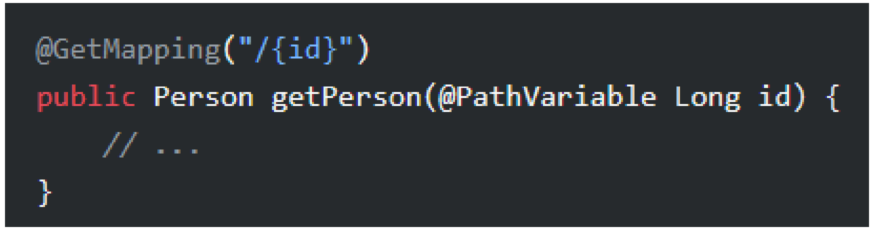
```
pages/
 ├── index.js          → /
 └── users/
      ├── index.js     → /users
      └── [id].js      → /users/:id
```

Routing: Next.js



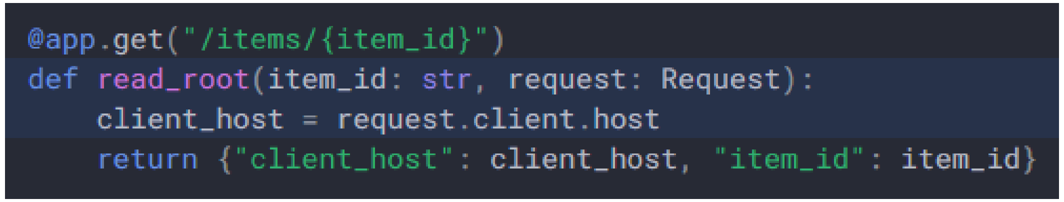
```
app.get('/', (req, res) => {
  res.send('hello world')
})
```

Routing: Express.js



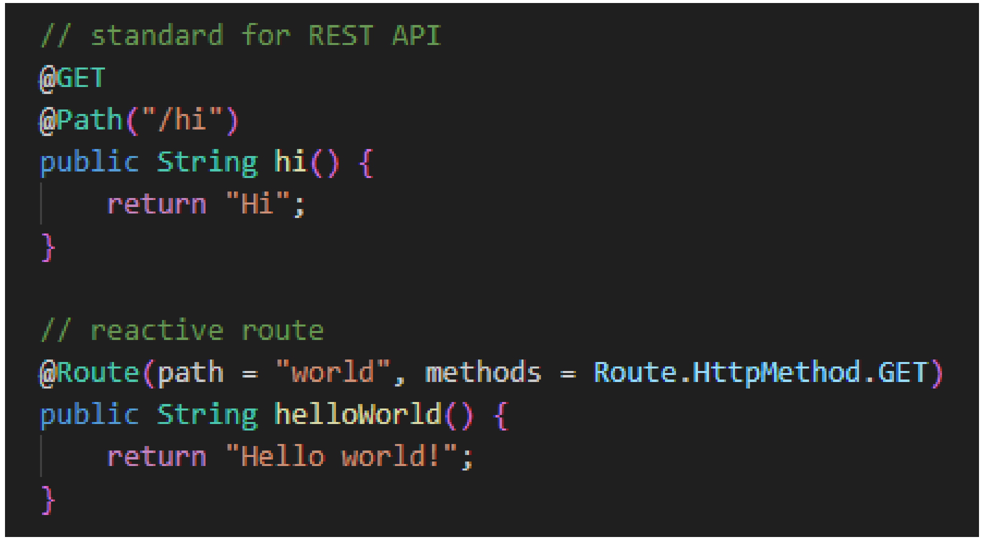
```
@GetMapping("/{id}")
public Person getPerson(@PathVariable Long id) {
    // ...
}
```

Routing: Spring Boot



```
@app.get("/items/{item_id}")
def read_root(item_id: str, request: Request):
    client_host = request.client.host
    return {"client_host": client_host, "item_id": item_id}
```

Routing: FastAPI



```
// standard for REST API
@GET
@Path("/hi")
public String hi() {
    return "Hi";
}

// reactive route
@Route(path = "world", methods = Route.HttpMethod.GET)
public String helloWorld() {
    return "Hello world!";
}
```

Routing: Quarkus

Middleware (API)

API middleware represents a software layer between API client and server, managing the requests and responses and accompanying tasks like authentication, logging and error handling and parsing.

Validation

Express.js needs a manual validation via third-party middleware wrapper like express-validator.

Spring Boot has annotations like @NotNull which are checked when the called method's parameter has a @Valid annotation.

FastAPI has automatic validation via pydantic models.

Next.js uses validation libraries for more advanced validation.

Quarkus, like Spring Boot, manages validation via annotations.

Authentication

Express.js uses middleware authentication.

Spring Boot supports a number of ways to authenticate oneself such as OAuth2, SAML 2.0 and CAS

FastAPI uses primarily OAuth2 today.

Next.js uses credentials and cookies.

Quarkus, like Spring Boot, supports a number of authentication mechanisms like OAuth2 and JWT

Concept of Integrating External / Describing REST-API

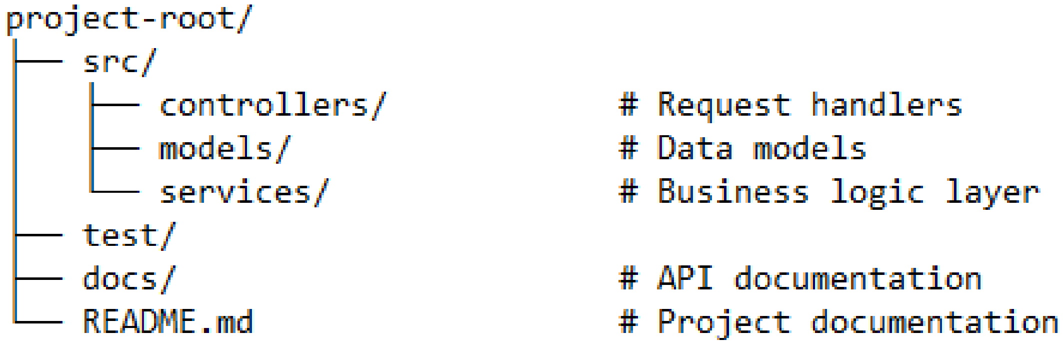
Integrating external REST-APIs describes using another systems API to handle data. For example: This could be used to process the payment of an Amazon order via the external REST-API of PayPal.

Describing REST-APIs focuses on documenting one's own API in a standardized, machine-readable format. This allows others integrating the API into their own system themself. This can be done manually or with frameworks like OpenAPI which can auto-generate the documentation from code annotations.

Typical Project Structure and recommended Design Pattern

A typical project structure would be a layered architecture as it allows for a simple seperation of concerns in:

- Controllers (handles HTTP-Requests)
- Services (backend logic)
- Models (database interactions)



```
project-root/
├── src/
│   ├── controllers/          # Request handlers
│   ├── models/               # Data models
│   └── services/             # Business logic layer
├── test/
├── docs/                     # API documentation
└── README.md                 # Project documentation
```

Figure: A typical project structure with layered architecture

Recommended Design Patterns

Publish-Subscribe for AAS-Data Synchronization:

- Publisher sends data to a broker where it is collected and distributed to Subscribers
- Asynchronous communication
- Dynamically scalable
- Components are decoupled

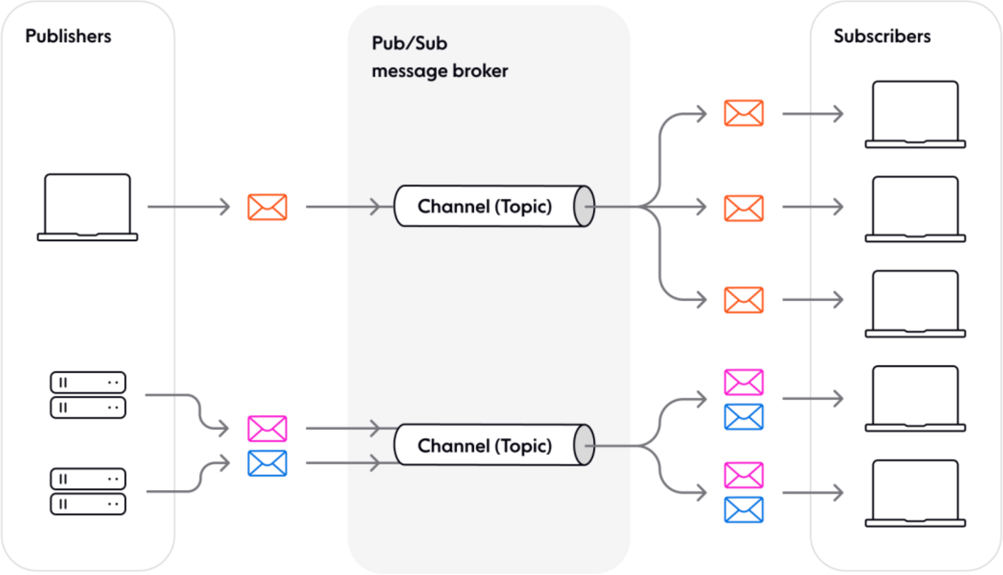


Figure: Publish-Subscribe, source

Recommended Design Patterns

Specification-Pattern for EDC Access Control:

- Allows for attribute-based access control
- Filters the results based on their current policy;
  as such if the policy demands the user to have the engineer attribute, only those with that attribute can see the respective results
- Policies can be composed of multiple AND, OR and NOT rules

Security Measures and best practices

- Use of authentication mechanisms, such as OAuth2 or JSON Web Tokens (JWT)
- Authorize user to restrict/grant access to only certain resources or actions
- Use of protected communication channels like HTTPS to encrypt data in transit
- Validate all input data to prevent injection attacks
- Limit and throttle the amount of requests a client can make in a certain time frame to prevent DoS attacks and ensure fair resource allocation
- Error handling and detailed logging to identify potential security risks
- Secure storage of credentials and secrets
- Regular security testing and audits

Sources

- https://www.ibm.com/docs/en/control-desk/7.6.1.2?topic=api-rest-framework
- https://www.orientsoftware.com/blog/api-frameworks/
- https://bell-sw.com/blog/what-is-quarkus/
- https://medium.com/codex/when-to-use-quarkus-over-spring-boot-and-how-to-get-started-1-ee546353d04c
- https://www.goalto.io/blog/nextjs-uses-features-examples
- https://leobit.com/blog/overview-of-next-js-for-modern-web-apps-pros-cons-and-use-cases/
- https://17vidushigupta.medium.com/real-time-use-cases-of-express-js-12a1bc53205a
- https://www.altexsoft.com/blog/expressjs-pros-and-cons/
- https://quarkus.io/guides/reactive-routes
- https://nextjs.org/docs/pages/building-your-application/routing/api-routes
- https://stoplight.io/api-documentation-guide
- https://www.geeksforgeeks.org/how-to-implement-validation-in-express-js/
- https://dev.to/taranka/validation-in-fastapi-3gam
- https://nextjs.org/docs/pages/building-your-application/data-fetching/forms-and-mutations
- https://www.geeksforgeeks.org/authentication-strategies-available-in-express/
- https://docs.spring.io/spring-security/reference/servlet/authentication/index.html
- https://fastapi.tiangolo.com/tutorial/security/
- https://nextjs.org/docs/pages/guides/authentication#session-management
- https://quarkus.io/guides/security-authentication-mechanisms
- https://ably.com/topic/pub-sub
- https://www.practical-devsecops.com/what-is-rest-api-security/