II. Implementation and DevOps Practices

4. Backend Development

This section chapter will …

The backend is the single source of truth in the form of a REST API which the frontend can interact with. There are two main areas which are backend handle

1. Business logic As with any logic handling money or other sensitive data, these operations must be done in an environment where bad actors or clueless users cannot alter the result. Therefore, since we are developing a gambling site, the actual game logic and results are calculated, handled and stored only in the backend. Our backend has API endpoints for playing games for users such as the stateless coin-flip endpoint.

   Response simplified for brevity

```
{
    "actions": [
        {
            "name": "play",
            "method": "POST",
            "href": "/game/coin-flip",
            "type": "application/json",
            "fields": [
                {
                    "name": "choice",
                    "type": "TEXT",
                    "options": [
                        { "value": "HEADS", },
                        { "value": "TAILS", }
                    ]
                },
                {
                    "name": "betAmount",
                    "type": "NUMBER",
                }
            ]
        },
        {
            "name": "result",
            "method": "GET",
            "href": "/game/coin-flip/{id}",
            "type": "application/x-www-form-urlencoded"
        }
    ],
    "links": [ { "rel": [ "self" ], "href": "/game/coin-flip" } ]
}
```

If the above JSON did not give it away, we use hypermedia to alter and interact with the state of the backend. Exactly how will be discussed later, for now the most important aspect of the request is that we have two endpoints to interact with the coin-flip game. We can either

```
1. Play the game via the POST method, requiring two arguments: whether the user choose
2. Get the result of a specific coin-flip game.
```

As mentioned before, this game is stateless, meaning once a client POST's or rather plays the coin-flip game, the result of the game is immediately calculated, saved and an ID linking to the result is returned for the client to consume via the second action 'GET /game/coin-flip/{id}. An example finished game could be the following json

Response simplified for brevity

```
{
    "properties": {
        "betAmount": 100,
        "gameResult": "USER_LOSE",
        "userChoice": "HEADS",
        "id": 1,
        "result": "TAILS"
    },
    "links": [ { "rel": [ "self" ], "href": "/game/coin-flip/1" } ]
}
```

Here after sending a `POST` request with the body `{ "choice": "HEADS", "betAmount": 100 }`, the user lost as the result was "TAILS".

2. Users The second main area of our API is handling users and authentication. We decided, to challenge ourselves and roll our own authentication and JWT tokens, the exact specifics of the JWT tokens and how they are created and managed will be examined later. In short, if a user wants to play a game they must first be registered and authenticate with a username and password to retrieve a short lived *access-token* (JWT) and a long lived *refresh-token* (UUID). The *access-token* grants the user access to locked endpoints based on the user's assigned roles like playing the coin-flip game and the *refresh-token* allows a user to refresh the *access-token* before it expires to create a new *access-token*. This is to reduce the number of times a user must send their credentials to the backend for increased security. If the *access-token* expires the user must login again with credentials and the same is true for the *refresh-token*.

Quarkus Framework Implementation Our choice of programming language and then framework was not up for much debate. We all have some experience with Java and therefore it was an obvious choice. However, instead of selecting Spring as our web framework we chose Quarkus for variation. Quarkus alike Spring supports the Jakarta EE standards meaning many of the core principles of developing a web server is the same between them. We chose Quarkus over Spring for a mild challenge in something new but familiar.

Quarkus is a performant container first web application and microservice framework. For our needs we utilized the web server components for constructing an API to accomodate our gambling website's business logic and user management. The main benefits we have utilized from a web application framework such as Quarkus is the ease of adding native but tailored components. As an example, we will delve into JWT tokens and how we use them later. The other main benefit of such a framework is Jakarta bean validation and dependency injection. We use this liberally with the `@Inject` annotation for *injecting* a class into another without having to manually construct it or the like. Instead Quarkus takes care of the scope of the class via `@RequestScoped` or `@ApplicationScoped` among a few others to figure out when and how a new instance of a class will be created and injected.

```
@ApplicationScoped
public class AuthService {

    @Inject
    TokenService tokenService;

    @Inject
    UserService userService;

    //...
}
```

Another key aspect of using Quarkus is the ease of creating a web application, specifically selecting which endpoints or resources should be exposed with what method.

```
@Path("/auth")
@Produces(Siren4J.JSON_MEDIATYPE)
public class AuthResource {

    //...

    @GET
    @Path("/")
    public Entity getAuth() throws Siren4JException {
        //...
    }

    @POST
    @Path("/token")
    @Consumes(MediaType.APPLICATION_JSON)
    public Entity createToken(CreateTokenRequest credentials) {
        //...
    }

    @POST
    @Path("/token/refresh")
    @Consumes(MediaType.APPLICATION_JSON)
    public Entity refreshAccessToken(RefreshAccessTokenRequest request) {
        //...
    }
}
```

Simple annotations define

- the top level path for the class or relative function level path via the `@Path` annotation,
- the common return type for all the endpoints in the class via the `@Produces` annotation,
- the HTTP method of the function via the `@GET`, `@PUT`, `@POST`, etc. annotations,
- the accepted Content-Type for a method via the `@Consumes` annotation and
- many other handy and useful features.

REST API Design with Siren Hypermedia The backend architecture evolved from a basic Level 3 REST implementation utilizing standard HTTP methods (GET, POST, DELETE) to incorporate HATEOAS (Hypermedia as the Engine of Application State) capabilities. This enhancement enables dynamic API navigation through hypermedia controls, where clients discover available resources and operations by traversing links from the root endpoint rather than relying on predefined URL patterns. The hypermedia approach mirrors web browsing behavior, where navigation occurs through semantic controls rather than explicit URL construction. For example, rather than hardcoding paths to submit orders, clients can programmatically follow named links to execute operations. This design pattern eliminates the need for clients to maintain URL knowledge or external API documentation, as the server communicates available state transitions through hypermedia responses. The implementation effectively transforms HTTP interactions from direct URL manipulation to a state machine driven by discoverable, context-aware controls.

We chose Siren as our hypermedia specification, although we also considered HAL and HAL-FORMS, but found HAL to be too primitive and HAL-FORMS to have insufficient support in Quarkus. Siren is an extension of the JSON format, `application/vnd.siren+json`. It has 5 main components.

1. Properties

The `properties` keyword is an optional reserved keyword which alike in a traditional JSON response describes the state of an entity.

```
{
    "properties": {
        "id": 1,
        "username": "admin"
    }
}
```

2. Links

   The `links` keyword is a required keyword. A staple of hypermedia it enable the discoverability and relational nature of APIs. A given resource is required to contain the `self` relation with an `href` to the very URL or resource the client is viewing. Other common relations are, for pagination, "prev" or "next", as well as other closely related resources.

```
{
    "links": [
        { "rel": [ "self" ], "href": "/users/1" }
    ]
}
```

3. Entities

   The `entities` keyword is optional but usually used for collections. For example the endpoint `/users` probably returns a collection of all the users in the system. An API architect can decide to embed the full resource or a partial representation of it. A client may then see the full resource representation by navigating via the entity's "self" `rel`.

```
{
    "entities": [
        {
            "class": [ "user" ],
            "rel": [ "item" ],
            "properties": {
                "id": 1,
                "username": "admin"
            },
            "links": [ { "rel": [ "self" ], "href": "/users/1" } ]
        },
        {
            "class": [ "user" ],
            "rel": [ "item" ],
            "properties": {
                "id": 2,
                "username": "user"
            },
            "links": [ { "rel": [ "self" ], "href": "/users/2" } ]
        }
    ]
}
```

4. Actions

   The `actions` keyword is an optional list of all related actions to the retrieved resource. Usually reserved for more advanced request like path templated GET requests `/users/{id}` or POST and PUT methods which may require specific contents in their request bodies. Actions were the missing piece for us when contemplating the incorporation of HAL. Without them a hypermedia API in our opinion is inadequate.

```
{
    "actions": [
        {
            "name": "create-user",
            "method": "POST",
            "href": "/users",
            "title": "Create user",
            "type": "application/json",
            "fields": [
                {
                    "name": "username",
                    "title": "Username",
                    "type": "TEXT",
                    "required": true
                },
                {
                    "name": "password",
                    "title": "Plain-text password",
                    "type": "TEXT",
                    "required": true
                }
            ]
        }
    ]
}
```

   The resulting generic request for an action would be

```
curl -X '{action method}' \
    -H "Content-Type: {action type}" \
    -d '{
        # for each field in the action:
        "{field name}": "<value>"
    }' \
    {action href}
```

For the specific “create-user” action above the request would be

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{
      "username": "JohnDoe",
      "password": "password1234"
    }' \
    /users
```

5. Class

   The final main component of the Siren specification is the optional `class` keyword. It describes the nature of an entity’s content, usually it is used as a descriptor for the returned resource.

   See the previous `entities` component for an example.

We chose a third party library for our Siren implementation. Unlike Spring with its format agnostic `WebMvcLinkBuilder`, Quarkus lacks a mature and cohesive hypermedia builder, hence we selected what we believe to be the most intuitive and feature rich Java Siren library, Siren4J. This library contains both a builder like the Spring WebMvcLinkBuilder for more custom runtime responses but so far we have only used the resource API through annotations.

A simple resource like our root `/` resource contains discoverable links to other areas and resources of our API.

```
@Siren4JEntity(entityClass = "root", uri = "/", links = {
    @Siren4JLink(rel = "users", href = "/users"),
    @Siren4JLink(rel = "auth", href = "/auth"),
    @Siren4JLink(rel = "game", href = "/game")
})
public class GetRootResponse {
}
```

Notice we link to three resources: “users”, “auth” and “game”, each with their own path “/users” “/auth” and “/game”. To properly serialize this response we use Siren4Js - admittedly slow - `ReflectingConverter` to convert the Java object into a Siren `Entity`.

```
@ApplicationScoped
public class RootController {

    public Entity getRoot() throws Siren4JException {
        GetRootResponse rootResponse = new GetRootResponse();
        return ReflectingConverter.newInstance().toEntity(rootResponse);
    }
}
```

And finally our `RootResource` which defines our endpoint simply calls the controller

```
@Path("/")
@Produces(Siren4J.JSON_MEDIATYPE)
public class RootResource {

    @Inject
    RootController rootController;

    @GET
    public Entity getRoot() throws Siren4JException {
        return this.rootController.getRoot();
    }
}
```

Performing a GET request on the `http://localhost:8080/` results in the following Siren JSON

```
{
    "class": [ "root" ],
    "properties": {
        "$siren4j.class": "org.csdg8.root.dto.GetRootResponse"
    },
    "links": [
        { "rel": [ "self" ], "href": "/" },
        { "rel": [ "users" ], "href": "/users" },
        { "rel": [ "auth" ], "href": "/auth" },
        { "rel": [ "game" ], "href": "/game" }
    ]
}
```

Notice the `class` keyword defined is the `entityClass` value "root" and the links are created one to one as they are defined in the annotation of the `GetRootResponse` class.

Siren4J automatically creates the `$siren4j.class` keyword in the response's `properties` and populates it with the full class path.

This same strategy of construction annotations on a DTO are used throughout our API on all endpoints. A limitation of this is that we must be careful when changing links as we have to hard-code the value in the annotations since Java annotations can only contain compile-time values. Although, as we will see later, tests help us in properly verifying that links match. If we needed a response with a runtime link or another runtime value we can utilize Siren4J's `LinkBuilder`. The following is an example from the library's GitHub page:

```
EXAMPLE BUILDER:

// Create a new self Link
Link selfLink = LinkBuilder.newInstance()
        .setRelationship(Link.RELATIONSHIP_SELF)
        .setHref("/self/link")
        .build();

// Create a new Entity
Entity result = EntityBuilder.newInstance()
        .setEntityClass("test")
        .addProperty("foo", "hello")
        .addProperty("number", 1)
        .addLink(selfLink)
        .build();
```

Database Integration *hibernate ORM and postgresql*

Business Logic Implementation *business logic separation, service classes*

Security Implementation *quarkus security with rolesallowed, 401 vs 403.*

JWT Authentication *jwt tokens, upn, claims etc. shortlived/longlived tokens what and why etc.*

Backend Security Measures *probably covered above*

Testing Strategy

JUnit Implementation

REST-assured Testing

OpenAPI Documentation