UNIVERSITÉ DE LORRAINE

Documentation technique
Groupe 2 : Surveys

Membres :

CONRAD Alexandre

DAVIES Liam

CHEVALET Clément

BIR Vincent

Liens :

https://gitlab.cedricmtta.com/Alexandre/surveys/-/tree/master/

https://sonarqube.cedricmtta.com/dashboard?id=fr.univ-lorraine.m2.gi%3Agroupe-2

Technologies utilisées :

Table des matières

Utilisation de Swagger

Nous avons utilisé Swagger puisque c'était l'occasion d'approfondir les notions vues en cours. Cela nous a permis de créer entièrement la structure du projet (création de l'API et des différents modèles). Cela nous a pris plusieurs jours, il fallait absolument réfléchir correctement au découpage des routes de l'API, des fonctions et des différents modèles qui allaient être mappés avec la base de données.

Notre ressenti est que cela nous a permis de gagner du temps dans la structure et au déploiement, mais aussi de réfléchir à comment organiser notre projet. En contrepartie, une génération automatique nous a donné beaucoup de code smells, de redondances de codes et des modèles ou classes inutiles. De ce fait, un travail sur le code a été nécessaire.

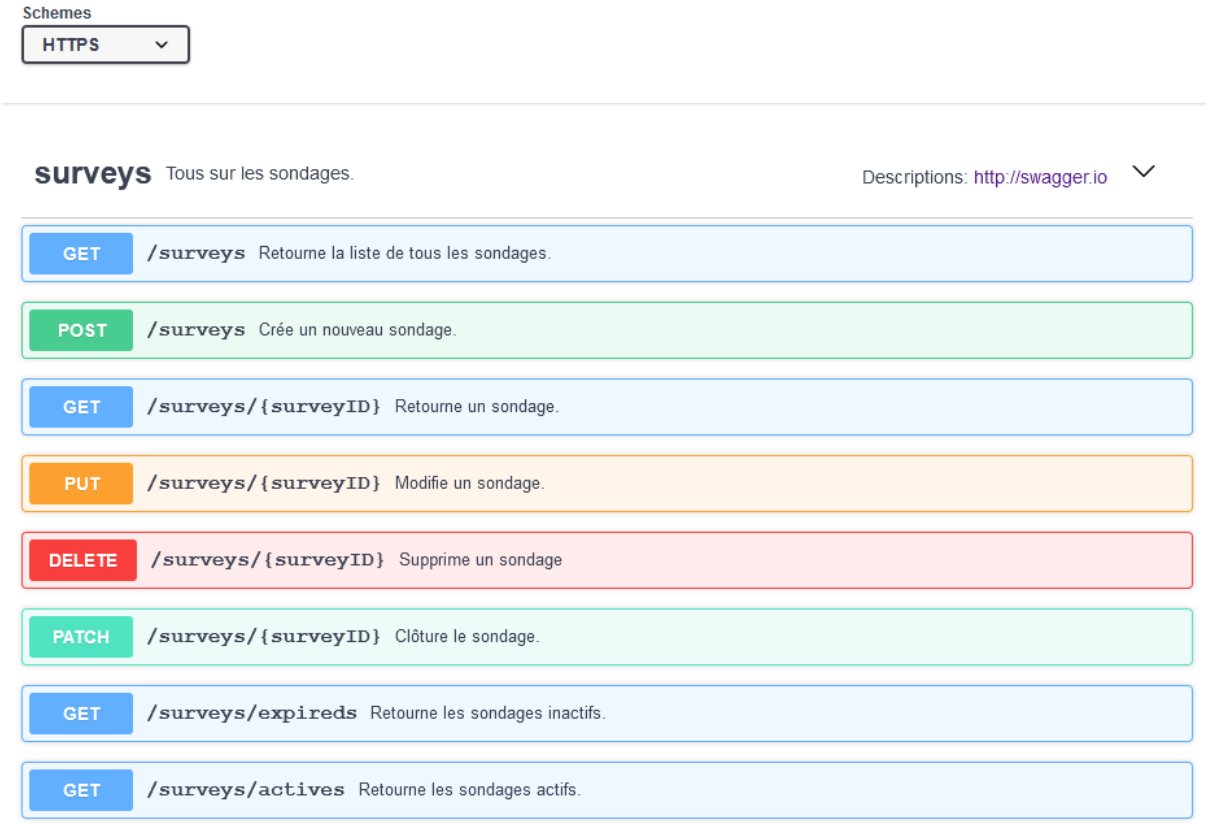


Figure 1: Documentation Swagger

Hibernate et H2

Notre base de données s'appuie sur H2, et nous utilisons Hibernate pour pouvoir communiquer avec la base de données.

De ce fait, nous avons essayé d'implémenter un modèle par couche comme celui vu en cours.

H2 est une base de données en mémoire, mais nous avons décidé de stocker les données dans un fichier. Au chargement de notre application, un import SQL charge un jeu de données dans la base de données. Tous les membres du groupe n'avaient jamais étudié H2, de ce fait nous avons décidé de l'utiliser pour le découvrir.

Nous avons choisi Hibernate comme pour H2 car nous l'avions aussi vu en cours et que personne ne l'avait utilisé auparavant. Ce framework nous a servi à mapper directement la base de données en classe java. Ceci nous a permis d'utiliser directement les classes Java comme des objets de la base de données. Un avantage d'Hibernate est que nous pouvons à tout moment changer la base donnée de H2 à MariaDB par exemple, à condition de respecter le mapping.

De plus, nous avons décidé d'aller un peu plus loin avec le mapping grâce à l'aide de notions que nous avions vu en cours, comme le OneToMany et le ManyToOne.

```
<!-- Mappage avec les classes de modèle contenant des annotations  -->
<mapping class="io.swagger.model.Survey"/>
<mapping class="io.swagger.model.Comment"/>
<mapping class="io.swagger.model.Choice"/>
<mapping class="io.swagger.model.Option"/>
<mapping class="io.swagger.model.Vote"/>
```

Figure 2: Le mapping Hibernate

```
@JsonProperty("isAvailable")
@NonNull
@Column(name = "IS_AVAILABLE")
Boolean isAvailable;

@JsonProperty("endDate")
@NonNull
@Column(name = "END_DATE")
Timestamp endDate;

@OneToMany(targetEntity = Choice.class, fetch = FetchType.LAZY, mappedBy = "idSurvey")
@ApiModelProperty(hidden = true)
@JsonIgnore
List<Choice> choices;
```

Figure 3: Exemple de mapping d'un model

Utilisation de Lombok

Lombok nous a permis de retirer énormément de lignes de code et d'accesseur. De ce fait, à l'aide de certaines annotations, nous avons pu générer automatiquement énormément de fonctions telles que : getter, setter, constructor, toString, hashCode.

Par exemple, d'après la capture ci-dessous, nous voyons qu'avec @Data nous avons pu nous dispenser d'écrire le getter, setter, troString et le hashCode.

```
/**
 * Survey
 */

/**
 * Lombok
 */
@Validated
//@javax.annotation.Generated(value = "io.swagger.codegen.languages.SpringCodegen", date = "2020-10-31T12:55:18.203Z")
@FieldDefaults(level = AccessLevel.PRIVATE)
@RequiredArgsConstructor // Constructeur par défaut impossible
@AllArgsConstructor
@NoArgsConstructor
@Data // annotation is the combination of @ToString, @EqualsAndHashCode, @Getter and @Setter.
```

Figure 4: Les annotations de la classe Survey

Nous avons ensuite énormément utilisé Lombok pour simplifier le model. Sur les captures ci-dessous, on observe que nous avons pu éviter toutes les déclarations des différents attributs.

```
public class Survey   {
  @JsonProperty("id")
  private Long id = null;

  @JsonProperty("name")
  private String name = null;

  @JsonProperty("description")
  private String description = null;

  @JsonProperty("isAvailable")
  private Boolean isAvailable = false;

  @JsonProperty("endDate")
  private OffsetDateTime endDate = null;

  @JsonProperty("comments")
  @Valid
  private List<SurveyComments> comments = null;

  @JsonProperty("votes")
  @Valid
  private List<Choice> votes = null;

  public Survey id(Long id) {
    this.id = id;
    return this;
  }
```

```
/** Hibernate*/
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"
public class Survey implements Serializable {

    @JsonProperty("idSurvey")
    @ApiModelProperty(hidden = true)
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_SURVEY")
    Long idSurvey;

    @JsonProperty("name")
    @NonNull
    @Column(name = "NAME")
    String name;

    @JsonProperty("description")
    @NonNull
    @Column(name = "DESCRIPTION")
    String description;

    @JsonProperty("isAvailable")
    @NonNull
    @Column(name = "IS_AVAILABLE")
    Boolean isAvailable;

    @JsonProperty("endDate")
    @NonNull
    @Column(name = "END_DATE")
    Timestamp endDate;

    @OneToMany(targetEntity = Choice.class, fetch = FetchTyp
    @ApiModelProperty(hidden = true)
    @JsonIgnore
    List<Choice> choices;

}
```

Gestion des exceptions

Lors de la création du Swagger Editor, nous avons réfléchi à l'arborescence et structure du projet mais aussi aux différents codes d'erreurs retournés selon les attributs envoyés.

En outre, SwaggerEditor nous a généré un premier template d'Exception au niveau de l'API et nous n'avions plus qu'à mettre en place la gestion des exceptions et des différents tests afin de retourner les bons codes d'erreur.

```
@ApiResponses(value = {
        @ApiResponse(code = 200, message = "opération réussie", response = Comment.class),
        @ApiResponse(code = 400, message = "Manque des informations dans le corps."),
        @ApiResponse(code = 409, message = "Certaines informations ne respectent pas les conditions."),
        @ApiResponse(code = 500, message = "Echec de connexion à la base de données.")})
```

Figure 5: Exemple de code d'utilisation

Les tests unitaires

Pour les tests unitaires, nous avons utilisé JUnit et Mockito. Ces deux systèmes d'analyse de code ont été étudiés en cours mais pratiquement jamais utilisés par les membres du groupe. La capture d'écran ci-dessous nous montre la réalisation d'un test à l'aide de JUnit.

```java
/**
 * Resolver pour les Analytics
 */
@Test
public void AvailableIOptionResolver(){

    /** Variables **/
    AvailableIOptionResolver maybe = new AvailableIOptionResolver();
    Option optDispo = new Option( name: "Disponible");
    Option optIndis = new Option( name: "Indisponible");

    /** Tests **/
    Assert.assertTrue(maybe.shouldMatch(optDispo));
    Assert.assertFalse(maybe.shouldMatch(optIndis));
}
```

Les tests plus élaborés avec Mockito permettent de tester les status de retour aux appels API.

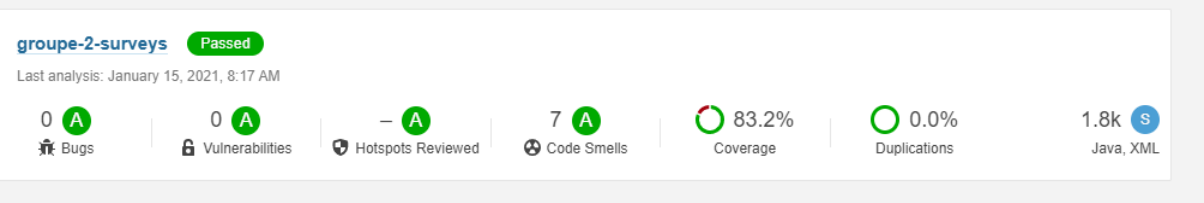
Avec Mockito nous avons pu aussi « mocker » des classes afin de les tester.

```java
@RunWith(MockitoJUnitRunner.class)
@ExtendWith(MockitoExtension.class)
@SpringBootTest
public class AnalyticsControllerTest {

    @Before
    public void init () { MockitoAnnotations.initMocks( testClass: this); }

    @Mock
    ObjectMapper objectMapper;

    @Mock
    HttpServletRequest httpServletRequest;

    @Mock
    HttpServletRequest httpServletRequestAccept;
```

SonarQube

SonarQube permet d'évaluer le coverage, les code smells et les redondances de tests dans un projet de manière continue. Il nous permet de voir quel code n'a pas de coverage et les bugs de différentes natures (sécurité, …).

De plus, il nous a servi aussi d'objectif grâce à la quality gate qui nous permet de montrer à quel niveau le code est considéré comme « réussi ».



Jacoco

Nous avions fait énormément de tests unitaires, mais aucun coverage n'était appliqué sur le projet. En cherchant un moyen pour faire lire nos tests par SonarQube, nous sommes tombés sur Jacoco qui permet simplement de créer un rapport. Il s'occupe de réaliser nos tests, puis génère un rapport et donne une note à chaque test. SonarQube n'a plus qu'à le lire afin d'attribuer la note sur le site.

Gitlab et Gitea

Nous avons utilisé Gitlab proposé par le cours afin de poster nos différents codes en respectant les différentes notions vues en cours en rapport à l'utilisation de git. Nous avons donc créé différentes branches afin de travailler chacun sur nos parties, avec une certaine convention.

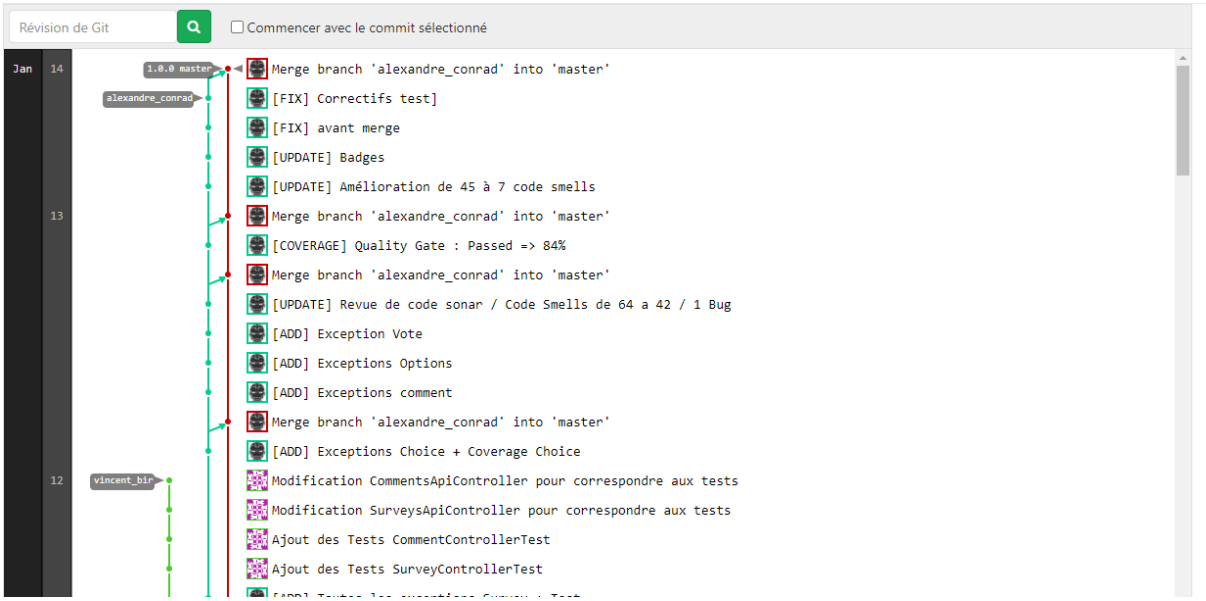


Figure 6: Exemple de commits sur Gitlab

Readme : Nous avons essayé de faire un Readme qui respecte les conventions de gros projets en mettant dedans les différents pré requis, les technologies utilisées, la documentation et l'installation ainsi que la version.

README.md

Surveys

Badges :

quality gate failed | reliability A | maintainability A | pipeline passed | coverage 66.5% | Chat Slack | Contributeurs 4
Le meilleur projet ? Oui , évidemment | code smells 54 | duplicated lines 0%

Projet de production logiciel réalisé par Monsieur Cédric Moschetta pour les Master 2 Génie informatique 2020/2021. L'objectif du projet est de réaliser une API pour des sondages en ligne. Le programme doit respecter certaines spécificités techniques comme un tunnel d'intégration, Sping Boot, etc.

Pré-requis

Pour utiliser notre projet, il suffit d'avoir la liste des applications ou logiciel ci-dessous:

- Maven - Pour les dépendances
- Java 11 - JRE JAVA

Installation

Pour utiliser notre API, il suffit :

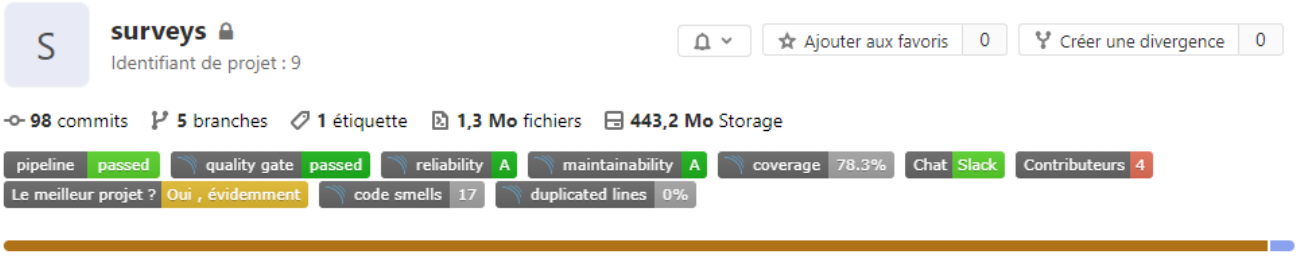
- API - Accès à l'interface de l'API
- PATH/PORT - Pour changer le PATH ou le port d'écoute

Démarrage

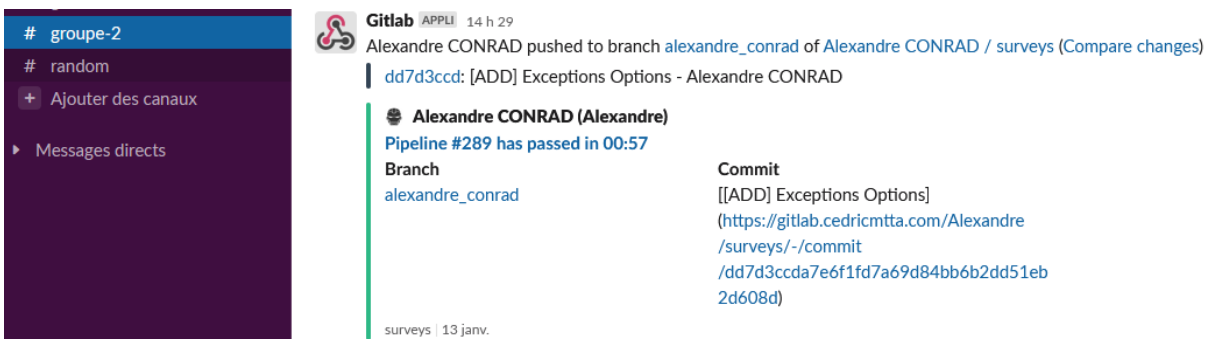
Afin de lancer le projet

- Documentation - Documentation
- H2 - Base de données

Badges : Nous avons utilisé des badges pour notre projet afin de récupérer les différentes notes et critères du SonarQube et d'autres technologies utilisées.



Webhook/Slack : Nous avons aussi utilisé un système de Webhook, qui nous a permis d'avoir une notification sur notre groupe à chaque fois qu'une personne pushait sur sa branche ou le master.



Les pipelines d'intégrations :

Afin de respecter les règles de Git et d'éviter de mettre en ligne un code qui ne fonctionne pas, un tunnel d'intégration a été mis en place pour valider le code. Si le code est validé (logo vert), le code peut être fusionné avec la branche master. Sinon l'utilisateur est dans l'incapacité de fusionner avec son projet. Un rapport est généré afin d'expliquer à l'utilisateur d'où vient son erreur afin de pouvoir la corriger.

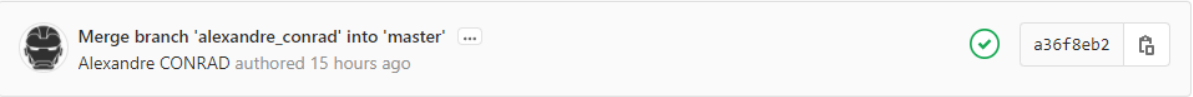


Figure 7: Exemple de pipeline d'intégration

Maven

A l'aide de Maven, nous avons créé un document « pom.xml » qui nous permet de gérer toutes les dépendances du projet, d'en rajouter ou d'en supprimer. Ce document nous a permis de rajouter toutes les technologies expliquées précédemment comme Hibernate, H2, Lombok.

```
<!-- Mockito pour les tests unitaires -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
</dependency>
<!-- H2 pour la data base -->
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.199</version>
    <scope>runtime</scope>
</dependency>
<!-- Hibernate pour le mapping -->
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.23.Final</version>
</dependency>
<!-- Lombok pour la gestion de getter/setter -->
```

Figure 8: Extrait du fichier pom (avec les imports nommés)