Spring DI in Vertx overview

Creating Spring context

```
public class InitAPIImpl implements InitAPI {
 @Override
 public void init(Vertx vertx, Context context, Handler<AsyncResult<Boolean>> handler) {
    vertx.executeBlocking(
      future -> {
        SpringContextUtil.init(vertx, context, ApplicationConfig.class);
        future.complete();
      },
      result -> {
        if (result.succeeded()) {
          handler.handle(Future.succeededFuture(true));
        } else {
          handler.handle(Future.failedFuture(result.cause()));
        }
      });
  }
}
```

We implemented class SpringContextUtil to use Spring dependency injection in Vert.x applications.

SpringContextUtil#init method initializes Spring context and adds it to Vertx Context object.

Accessing Spring context from endpoint classes

```
public class EholdingsProxyTypesImpl implements EholdingsProxyTypes {

 private final Logger logger = LoggerFactory.getLogger(EholdingsProxyTypesImpl.class);

 @Autowired
 private RMAPIConfigurationService configurationService;
 @Autowired
 private ProxyConverter converter;
 @Autowired
 private HeaderValidator headerValidator;

 public EholdingsProxyTypesImpl() {
  SpringContextUtil.autowireDependencies(this, Vertx.currentContext());
 }
```

SpringContextUtil#autowireDependencies method gets Spring context from Vertx context and uses it to inject beans into EholdingsProxyTypesImpl.

RestVerticle will call default constructor through reflection.

Declaring Spring configuration

SpringContextUtil#init uses Spring configuration class.

Detailed documentation on how to declare Spring configuration can be found here:

https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java

One of the approaches is to use @ComponentScan to automatically detect beans and add them to context:

```
@Configuration
@ComponentScan(basePackages = {
  "org.folio.rest.converter",
  "org.folio.rest.parser",
  "org.folio.rest.validator",
  "org.folio.http",
  "org.folio.config.impl",
  "org.folio.config.cache"})
public class ApplicationConfig {
  @Bean
 public PropertySourcesPlaceholderConfigurer placeholderConfigurer(){
    PropertySourcesPlaceholderConfigurer configurer = new PropertySourcesPlaceholderConfigurer();
    configurer.setLocation(new ClassPathResource("application.properties"));
    return configurer;
  }
}
```

basePackages parameter specifies list of packages that will be searched. Any class in those packages that is annotated with @Component will be added to context.

PropertySourcesPlaceholderConfigurer class allows to use properties from file application.properties on classpath.

Example of bean:

```
@Component
public class RMAPIConfigurationCache {

  private Vertx vertx;
  private long expirationTime;

 @Autowired
 public RMAPIConfigurationCache(Vertx vertx, @Value("${configuration.cache.expire}") long expirationTime) {
    this.vertx = vertx;
    this.expirationTime = expirationTime;
  }
```

vertx and expirationTime parameters will be automatically injected by Spring, expirationTime will be set to the value of "configuration.cache.expire" property from application.properties file.

Benefits of using Spring DI

1) Spring allows to inject properties by using @Value annotation, without this we need to manually read file and share it across application.

2) Vertx specific objects (e.g. Vertx, Context) can be injected into any bean, it is not necessary to pass them into methods as parameters.

3) It simplifies initialization of beans and decouples them from concrete implementation.

Old approach for creating object:

```
  private RMAPIConfigurationService configurationService;
  private PackagesConverter converter;
  private HeaderValidator headerValidator;
  private PackageParametersValidator packageParametersValidator;
  private PackagePutBodyValidator packagePutBodyValidator;
  private CustomPackagePutBodyValidator customPackagePutBodyValidator;
  private PackagesPostBodyValidator packagesPostBodyValidator;
  private TitleParametersValidator titleParametersValidator;
  private ResourcesConverter resourceConverter;
  private IdParser idParser;

  public EholdingsPackagesImpl() {
    this(
      new RMAPIConfigurationServiceCache(
        new RMAPIConfigurationServiceImpl(new ConfigurationClientProvider())),
      new HeaderValidator(),
      new PackageParametersValidator(),
      new PackagePutBodyValidator(),
      new CustomPackagePutBodyValidator(),
      new PackagesPostBodyValidator(),
      new PackagesConverter(),
      new TitleParametersValidator(),
      new ResourcesConverter(),
      new IdParser());
  }
  // Surpressed warning on number parameters greater than 7 for constructor
  @SuppressWarnings("squid:S00107")
  public EholdingsPackagesImpl(RMAPIConfigurationService configurationService,
                               HeaderValidator headerValidator,
                               PackageParametersValidator packageParametersValidator,
                               PackagePutBodyValidator packagePutBodyValidator,
                               CustomPackagePutBodyValidator customPackagePutBodyValidator,
                               PackagesPostBodyValidator packagesPostBodyValidator,
                               PackagesConverter converter,
                               TitleParametersValidator titleParametersValidator,
                               ResourcesConverter resourceConverter,
                               IdParser idParser) {
    this.configurationService = configurationService;
    this.headerValidator = headerValidator;
    this.packageParametersValidator = packageParametersValidator;
    this.packagesPostBodyValidator = packagesPostBodyValidator;
    this.converter = converter;
    this.packagePutBodyValidator = packagePutBodyValidator;
    this.customPackagePutBodyValidator = customPackagePutBodyValidator;
    this.titleParametersValidator = titleParametersValidator;
    this.resourceConverter = resourceConverter;
    this.idParser = idParser;
  }
```

With Spring:

```
  @Autowired
  private RMAPIConfigurationService configurationService;
  @Autowired
  private PackagesConverter converter;
  @Autowired
  private HeaderValidator headerValidator;
  @Autowired
  private PackageParametersValidator packageParametersValidator;
  @Autowired
  private PackagePutBodyValidator packagePutBodyValidator;
  @Autowired
  private CustomPackagePutBodyValidator customPackagePutBodyValidator;
  @Autowired
  private PackagesPostBodyValidator packagesPostBodyValidator;
  @Autowired
  private TitleParametersValidator titleParametersValidator;
  @Autowired
  private ResourcesConverter resourceConverter;
  @Autowired
  private IdParser idParser;

  public EholdingsPackagesImpl() {
    SpringContextUtil.autowireDependencies(this, Vertx.currentContext());

  }
```

4) It resolves circular dependencies between objects.

For example mod-kb-ebsco-java has a lot of converter classes that depend on each other. This makes it impossible to use approach with default constructor from above.

5) Overall, depency injection makes it easier to create new classes, which promotes smaller more testable classes.

6) Objects are created once instead of being created on each request.

Presentation on dependency injection Spring_DI.pptx