VERT.X

Asynchronous and Reactive Programming

MAIN POINTS

- What is Eclipse Vert.x?
- A Fast Prototype for a starter
- Improving the design by refactorings
  - Separation of concerns at Vertices level
  - Extracting Vert.x services
  - Junit tests for testing asynchronus code.
- Consumming && Exposing third party HTTP/Json services
- Going further

WHAT IS ECLIPSE VERT.X?

SECTION – I:

- Basics of Vert.x
  - Vert.x is not a Framework, but a toolkit. The core libraries define the fundamental APIS for writing asynchronous networked applications
  - Then you pick the other useful modules for your application
    - DB Connection
    - Monitoring
    - Authentication
    - Logging
    - Service Discovery
    - Clustering support
  - Vert.x is based on the Netty Project

WHICH BUILD ENV?

- No one in particular
- Or…The one you prefer
- Vert.x Core is a simple jar
  - It can be embedded inside apps packaged as a Set of jars
  - Or it can be deployed inside popular components and application containers

WHY USE VERT.X?

- Vert.x is designed for aynchronous communications
- It can deal with more concurrent network communications with less threads than synchronous APIs such as java servlet or java.net socket classes
- It is usefull for a a wide range of applications
  - ➔ High volume message /event processing
  - ➔ Micro services
  - ➔ API Gateways
  - ➔ http Apis for mobile applications
  - ➔ But also, … more traditional web apps (not limited to the heavy/complex or high demanding type of apps)

BUILDING WITH ASYNCHRONICITY IN MIND

- The Vert.x code does not look as complex as it sounds
- But if there is a sudden peak in traffic, then the code is already written with the essential ingredient for scaling up:

- *Asynchronous Processing of Events*

WHAT LANGUAGE DOES IT SPEAK?

- It‘s polyglot
  - Java
  - Groovy
  - Scala
  - Kotlin
  - Javascript
  - Ruby
  - Celon
- Also the language specific API are idiomatic in each target language (example. Using Scala futures in place of Vert.x futures)

Java | JavaScript | Groovy | Ruby | Ceylon | Scala | Kotlin

```java
import io.vertx.core.AbstractVerticle;
public class Server extends AbstractVerticle {
  public void start() {
    vertx.createHttpServer().requestHandler(req -> {
      req.response()
        .putHeader("content-type", "text/plain")
        .end("Hello from Vert.x!");
    }).listen(8080);
  }
}
```

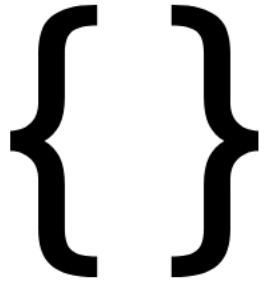
Polyglot

You can use Vert.x with multiple languages including **Java**, **JavaScript**, **Groovy**, **Ruby**, **Ceylon**, **Scala** and **Kotlin**.

Vert.x doesn't preach about what language is *best* — you choose the languages *you* want based on the task at hand and the skill-set of your team.

We provide *idiomatic* APIs for every language that Vert.x supports.

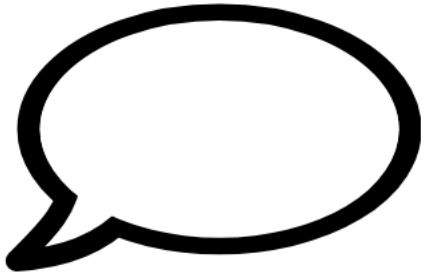
General purpose



Vert.x is incredibly flexible - whether it's simple network utilities, sophisticated modern web applications, HTTP/REST microservices, high volume event processing or a full blown back-end message-bus application, Vert.x is a great fit.

Vert.x is used by many different companies from real-time gaming to banking and everything in between.

Unopinionated

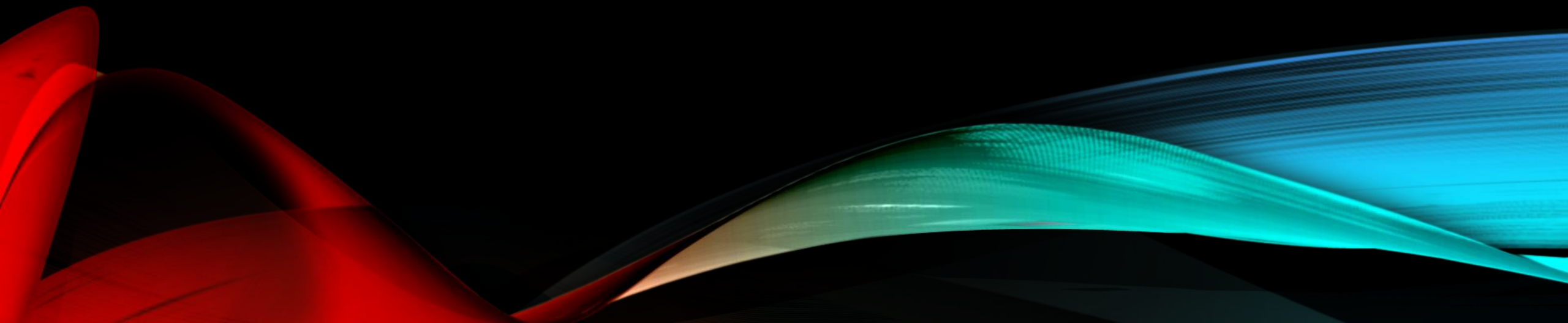


Vert.x is not a restrictive framework or container and we don't tell you a *correct* way to write an application. Instead we give you a lot of useful bricks and let you create your app the way *you* want to.

Need some guidance? We provide a large selection of examples to get you started for the particular type of application you want to write.

CORE CONCEPTS

Verticles and event Bus communication



VERTICLES AND THE BUS COMMUNICATION

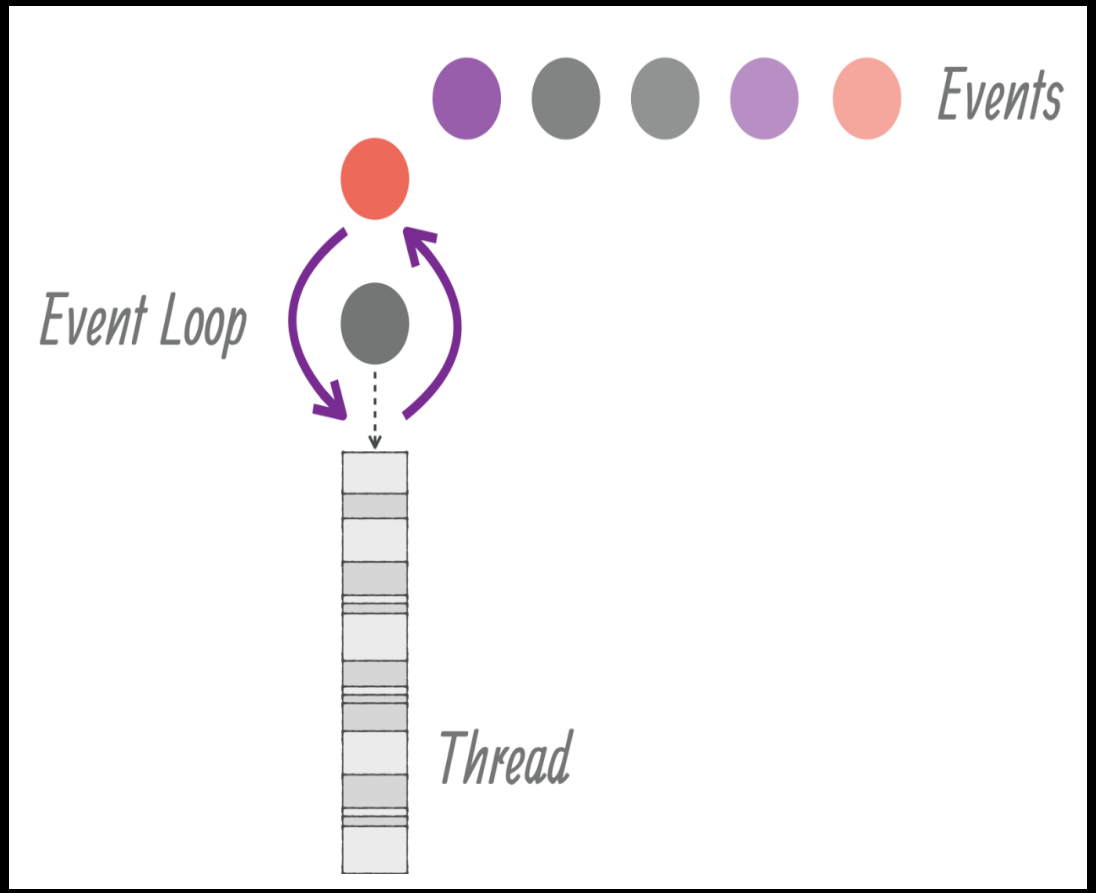
THREADING AND PROGRAMMING MODELS

- The traditional approach sucks by scalability !
  - One thread per network client
    - Assigned generaly at the establishment of the connection
    - Used for example with Servlets
    - Or network coding written with java.io and java.net
  - It‘s a „synchronous IO Threading model“
  - It looks simple as programming model
  - It sucks by scalability
    - Too many concurrent connection as threads are not cheap
    - Under heavy load, thread management and scheduling becomes itself heavy by the Operating System
  - Thus we need „asynchronous IO“ and Vert.x is here to help

VERTICLES

Transitioning to a better world!!!

VERTICLE: THE UNIT OF DEPLOYMENT



The unit of deployment is called a verticle.

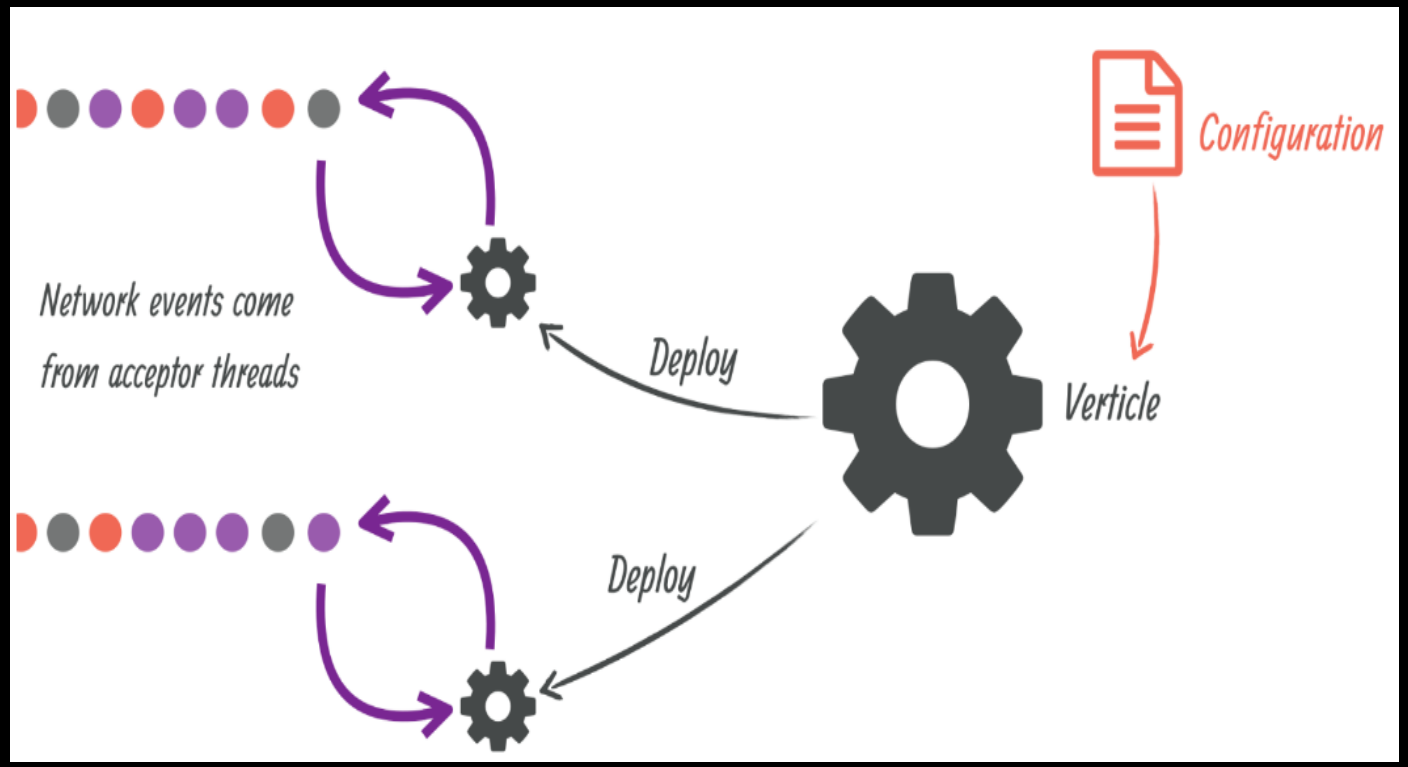
- A verticle processes incoming event over an event loop

- Events can be anything
  - → receiving network buffers
  - → timing events
  - → messages sent by other verticles

THE POLITICALY CORRECT EVENT HANDLING BY A VERTICLE

- Each event shall be processed in a **reasonable** amount of time to not block the event loop
- Therefore no thread blocking operation shall be performed while executed on the event loop
- Exactly like processing event in a graphical user interface (for example freezing a swing interface by executing a slow network operation)
- Vert.x offer mechanisms to deal with blocking operations outside the event loop

CHARACTERISTICS

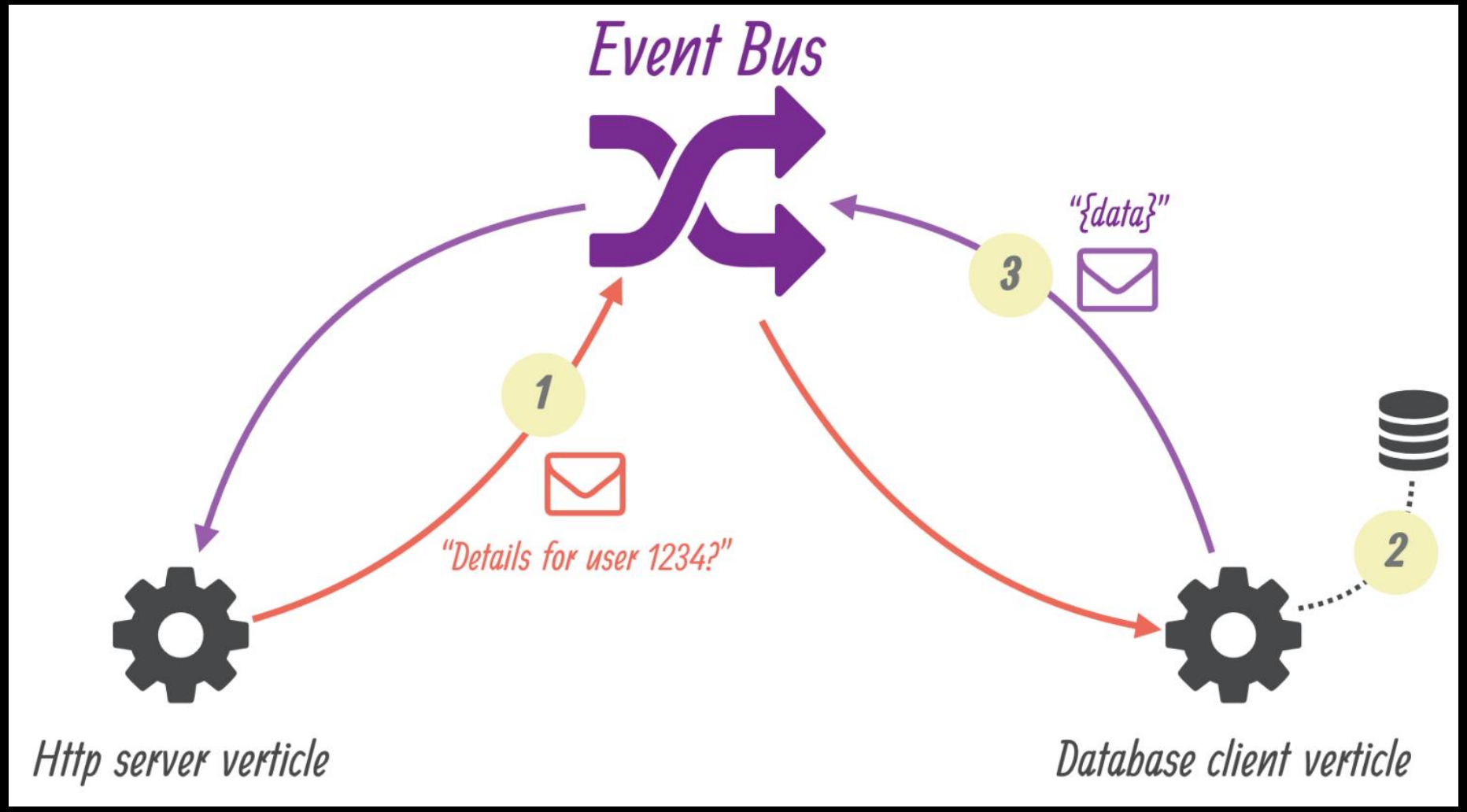


- Every event loop is attached to a thread
- By default Vert.x attaches two event loop per Core cpu thread
- As a consequence a regular verticle always process events on the same thread
- So there is no need for a thread cop
- A verticle can be passed some configurations: credentials, network adresse, etc
- A verticle can be deployed several times

CHARACTERISTICS

- Incoming network datas are being received from accepting threads and then passed as events to the corresponding verticles
- *When a verticle* opens a network server and **is deployed more than once**, then **the events are being distributed to the verticle instances** in a round-robin fashion which is very useful for maximizing CPU usage with lots of concurrent networked requests
- Verticles have a simple start/stop lifecycle
- Verticles can deploy other verticles

MESSAGE EXCHANGE THROUGH THE EVENT BUS



CHARACTERISTICS OF THE EVENT BUS

- All kinds of datas can transition between verticles through the event Bus
- Json is the preferred format because it is cross language.
- Messages can be sent to destinations which are ***free form strings***
- The event bus supports the following communication patterns
  - Point to point messaging
  - Request-response messaging
  - Publish/Subscribe for broadcasting messages

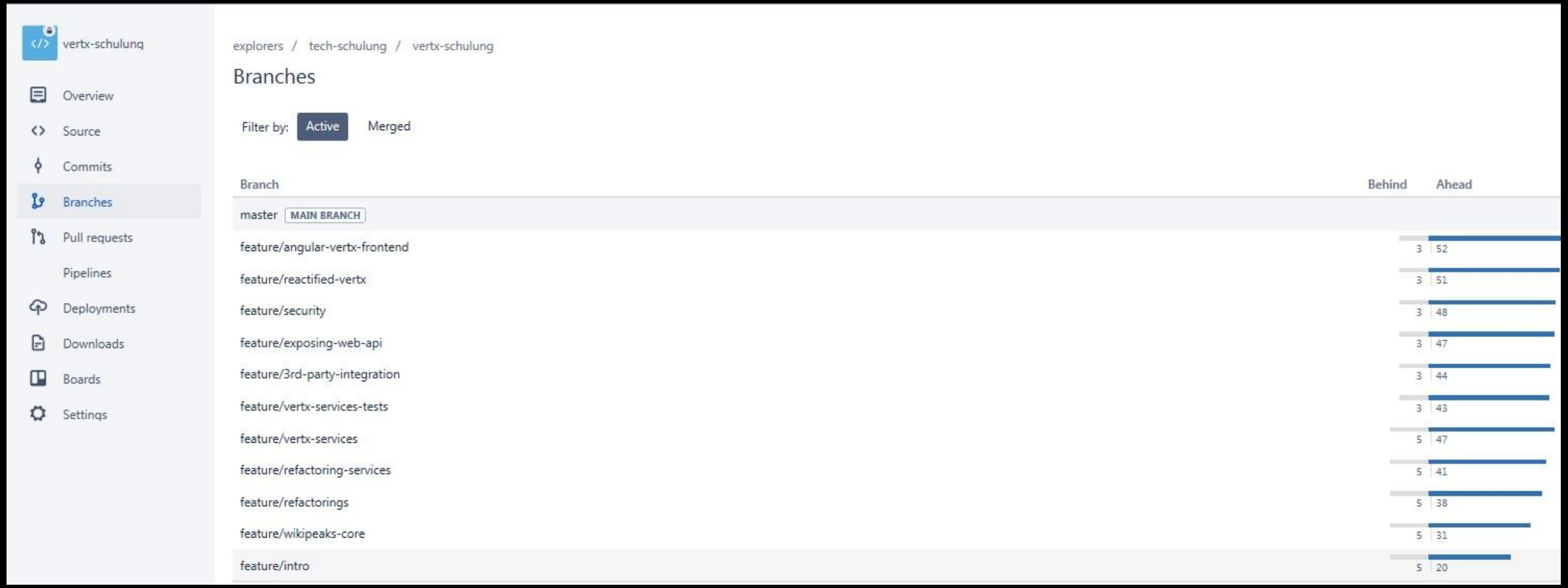
. . .

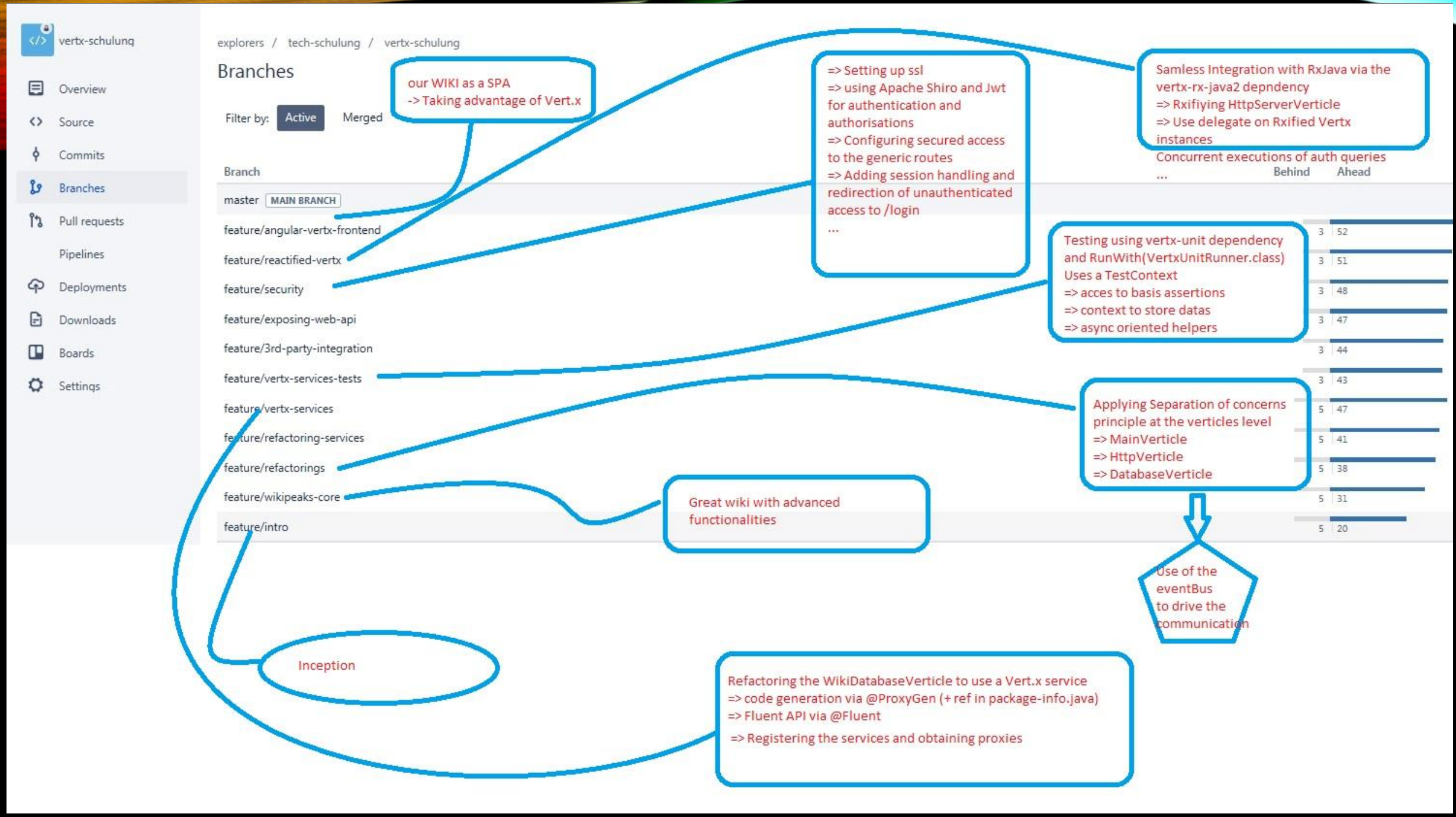
- Via the Event Bus, verticles can transparently communicate accros the same JVM or accross different JVMs
  - ***when network clustering is activated***, the event bus is *distributed* so that messages can be sent to verticles running on other application nodes,
  - Third party apps can communicate with each others via the Event Bus using a simple TPC protocol
  - The event bus can also be exposed over general purposes message bridge (example: AMQP, Stomp…)
  - a SockJS bridge allows web applications to seamlessly communicate over the event bus from JavaScript running in the browser by receiving and publishing messages just like any verticle would do.

FIRST DEMO

Wikipeaks

WIKIPEAKS BRANCHES




vertx-schulung
Overview
Source
Commits
Branches
Pull requests
Pipelines
Deployments
Downloads
Boards
Settings
explorers / tech-schulung / vertx-schulung
Branches
Filter by: Active Merged
Branch
Behind
Ahead
master MAIN BRANCH
feature/angular-vertx-frontend
feature/reactified-vertx
feature/security
feature/exposing-web-api
feature/3rd-party-integration
feature/vertx-services-tests
feature/vertx-services
feature/refactoring-services
feature/refactorings
feature/wikipeaks-core
feature/intro
3 52
3 51
3 48
3 47
3 44
3 43
5 47
5 41
5 38
5 31
5 20
our WIKI as a SPA
-> Taking advantage of Vert.x
=> Setting up ssl
=> using Apache Shiro and Jwt for authentication and authorisations
=> Configuring secured access to the generic routes
=> Adding session handling and redirection of unauthenticated access to /login
...
Samless Integration with RxJava via the vertx-rx-java2 depndency
=> Rxifiying HttpServerVerticle
=> Use delegate on Rxified Vertx instances
Concurrent executions of auth queries
...
Testing using vertx-unit dependency and RunWith(VertxUnitRunner.class)
Uses a TestContext
=> acces to basis assertions
=> context to store datas
=> async oriented helpers
Applying Separation of concerns principle at the verticles level
=> MainVerticle
=> HttpVerticle
=> DatabaseVerticle
Use of the eventBus to drive the communication
Great wiki with advanced functionalities
Inception
Refactoring the WikiDatabaseVerticle to use a Vert.x service
=> code generation via @ProxyGen (+ ref in package-info.java)
=> Fluent API via @Fluent
=> Registering the services and obtaining proxies

WHAT IS IT?

- A quick prototyping of a wiki app using Vert.x
- Server-side rendering of Html pages
- Data access through a JDBC connection
- Using the following librairies
- ➔ Vert.x web : provides elegant API to deal with routing, handling of request payloads, etc….
- ➔ Vert.x JDBC Client: provides an asynchronous API over JDBC
- ➔ Apache Free Marker: templating engine
- ➔ TxtMark: render markdown text as html allowing edition of wiki pages with markdown

- Bootstrapped maven project using: https://github.com/vert-x3/vertx-maven-starter
- Creating a fat jar with the help of Maven Shade Plugin

Anatomy of a verticle

The verticle for our wiki consists of a single `io.vertx.guides.wiki.MainVerticle` Java class. This class extends `io.vertx.core.AbstractVerticle`, the base class for verticles that mainly provides:

1. life-cycle `start` and `stop` methods to override,
2. a *protected* field called `vertx` that references the Vert.x environment where the verticle is being deployed,
3. an accessor to some configuration object that allows passing external configuration to a verticle.

To get started our verticle can just override the `start` method as follows:

```
public class MainVerticle extends AbstractVerticle {

  @Override
  public void start(Future<Void> startFuture) throws Exception {
    startFuture.complete();
  }
}
```

COMPOSITION OF VERT.X FUTURE

- Vert.x futures are not JDK futures.
- They can be composed and queried in a non-blocking fashion.
- They shall be used for simple coordination of asynchronous tasks, particularly for:
  - Deploying verticles
  - Checking if they are successfully deployed or not

- The Vert.x core APIs are based on callbacks to *notify of asynchronous events*

AVOID CALLBACK HELL TYPE OF PROGRAMMING

```
foo.a(1, res1 -> {
  if (res1.succeeded()) {
    bar.b("abc", 1, res2 -> {
      if (res.succeeded()) {
         baz.c(res3 -> {
           dosomething(res1, res2, res3, res4 -> {
               // (...)
           });
         });
      }
    });
  }
});
```

WIKIPEAKS VERTICLE INITIALISATION PHASE

1. We need to establish a JDBC db connection and make sure the db schema is in place
2. And we need to start a http server for the web app

- Each phase can fail (ex: http port is not available, …)
- They should not run in parallel as the web app needs the db access to work first

CODE STRUCTURE

```
private Future<Void> prepareDatabase() {
  Future<Void> future = Future.future();
  // (...)
  return future;
}

private Future<Void> startHttpServer() {
  Future<Void> future = Future.future();
  // (...)
  return future;
}
```

COMPOSITION IN THE START METHOD

By having each of our main process returns a Future, the start method becomes a composition

```
// tag::start[]
@Override
public void start(Future<Void> startFuture) throws Exception {
  Future<Void> steps = prepareDatabase().compose(v -> startHttpServer());
  steps.setHandler(startFuture.completer());
}
// end::start[]

public void anotherStart(Future<Void> startFuture) throws Exception {
  // tag::another-start[]
  Future<Void> steps = prepareDatabase().compose(v -> startHttpServer());
  steps.setHandler(ar -> {  // <1>
    if (ar.succeeded()) {
      startFuture.complete();
    } else {
      startFuture.fail(ar.cause());
    }
  });
  // end::another-start[]
}
```

WHAT HAPPENS IN START

- When the future of *prepareDatabase* completes successfully, then *startHttpServer* is called and the steps future completes depending on the outcome of the future returned by startHttpServer
- *startHttpServer* is never called if *prepareDatabase* encounters an error, in which case the *steps* future is in a failed state and becomes completed with the exceptions describing the error
- Eventually steps completes: *setHandler* defines a handler to be called upon completion. *In our case we simply want to complete startFuture* with steps and use the completer method to obtain a handler: the two start methods shown above are equivalent

USING VERT.X SERVICE

```java
@ProxyGen
public interface WikiDatabaseService {

  @Fluent
  WikiDatabaseService fetchAllPages(Handler<AsyncResult<JsonArray>> resultHandler)

  @Fluent
  WikiDatabaseService fetchPage(String name, Handler<AsyncResult<JsonObject>> resu

  @Fluent
  WikiDatabaseService createPage(String title, String markdown, Handler<AsyncResul

  @Fluent
  WikiDatabaseService savePage(int id, String markdown, Handler<AsyncResult<Void>>

  @Fluent
  WikiDatabaseService deletePage(int id, Handler<AsyncResult<Void>> resultHandler)

  // (...)
```

OBTAINING A SERVICE PROXY

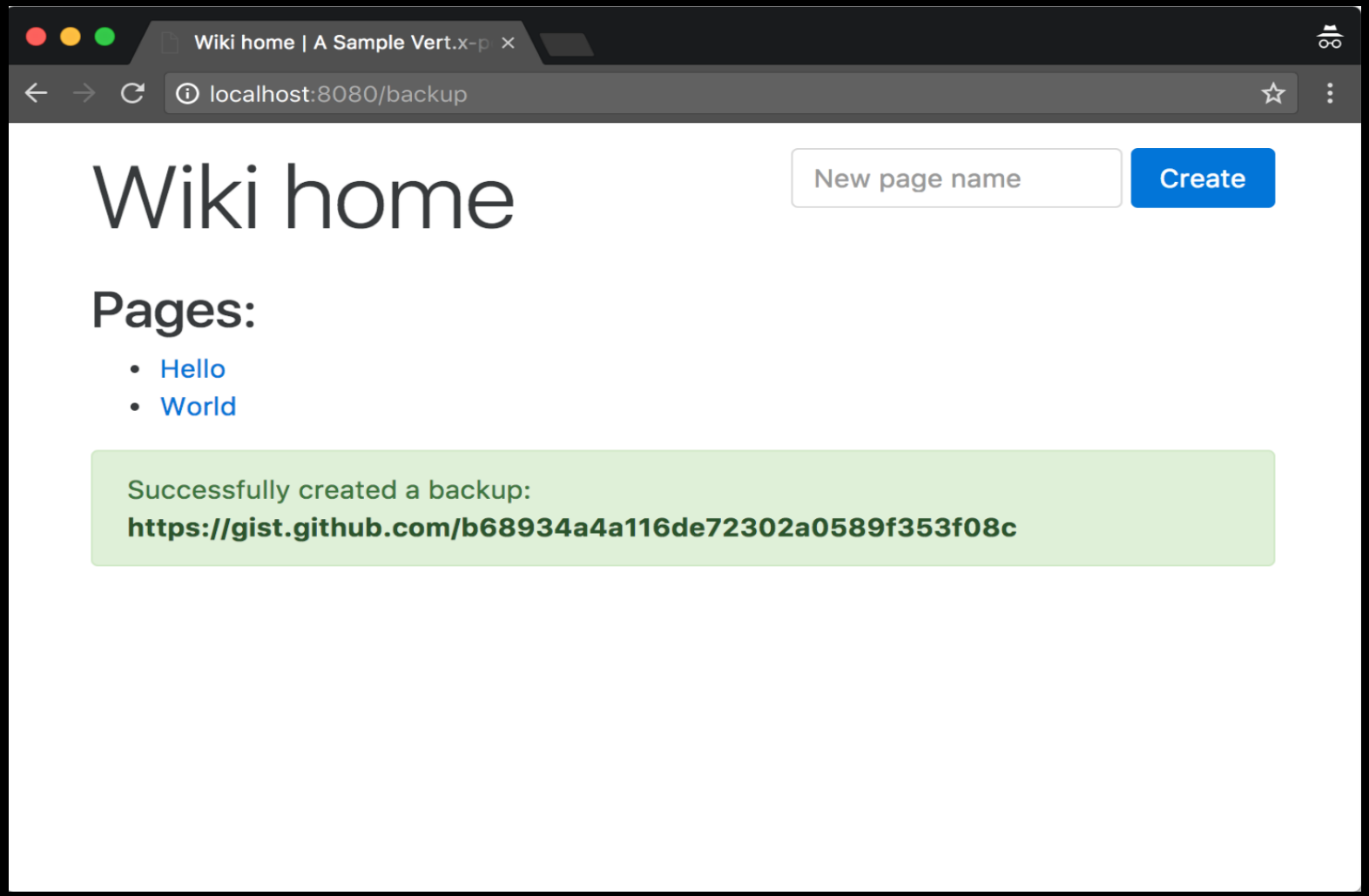
The Vert.x code generator creates the proxy class and names it by suffixing with `VertxEBProxy`. Constructors of these proxy classes need a reference to the Vert.x context as well as a destination address on the event bus:

```
static WikiDatabaseService createProxy(Vertx vertx, String address) {
  return new WikiDatabaseServiceVertxEBProxy(vertx, address);
}
```

TESTING WITH ASYNCHRONICITY IN MIND

```
@Test
public void crud_operations(TestContext context) {
  Async async = context.async();

  service.createPage("Test", "Some content", context.asyncAssertSuccess(v1 -> {

    service.fetchPage("Test", context.asyncAssertSuccess(json1 -> {
      context.assertTrue(json1.getBoolean("found"));
      context.assertTrue(json1.containsKey("id"));
      context.assertEquals("Some content", json1.getString("rawContent"));

      service.savePage(json1.getInteger("id"), "Yo!", context.asyncAssertSuccess(

        service.fetchAllPages(context.asyncAssertSuccess(array1 -> {
          context.assertEquals(1, array1.size());

          service.fetchPage("Test", context.asyncAssertSuccess(json2 -> {
            context.assertEquals("Yo!", json2.getString("rawContent"));

            service.deletePage(json1.getInteger("id"), v3 -> {

              service.fetchAllPages(context.asyncAssertSuccess(array2 -> {
                context.assertTrue(array2.isEmpty());
                async.complete();  (1)
              }));
```

INTEGRATING 3RD PARTY-API



SECURING

Authentication and Access Control

MAIN POINTS

- Moving from http to https
- Add user authentication with groups based priviledges
- Control access to the web api using Json Web Tokens (JWT)

GENERATING A SELF – SIGNED CERTIFICATE

```
keytool -genkey \
  -alias test \
  -keyalg RSA \
  -keystore server-keystore.jks \
  -keysize 2048 \
  -validity 360 \
  -dname CN=localhost \
  -keypass secret \
  -storepass secret
```

ACTIVATING SSL

- At the HttpServer creation we pass a HttpServerOptions object to specify that we want SSL and to point to our KeyStore file:

```
HttpServer server = vertx.createHttpServer(new HttpServerOptions()
  .setSsl(true)
  .setKeyStoreOptions(new JksOptions()
    .setPath("server-keystore.jks")
    .setPassword("secret")));
```

- By pointing to our sever (http://localhost:8080) we get the standard security exception common
- To most browsers since our certificate is self - signed

OPTIONS FOR AUTHENTICATION AND AUTHORISATIONS

- There is a wide range of options for that with Vert.x
- => JDBC, MongoDB, Apache Shiro, OAuht2,
  - With well known providers
  - Or with Jwt

BASICS USERS/PERMISSIONS/ROLES CONFIG

wiki-users.properties

```
# A user prefixed entry is a user account where the first value entry is the password: It may be followed by roles.
# for example the user: [root] has a password [w00t] and is an [admin]
# we use the role prefix to describe roles: an [admin] has all permissions. A [writer] has only the [update] permission
user.root=w00t,admin
user.foo=bar,editor,writer
user.bar=baz,writer
user.baz=baz

role.admin=*
role.editor=create,delete,update
role.writer=update
```

ADDING SECURITY TO THE HTTPSERVERVERTICLE

users.properties × | HttpServerVerticle.java × | PageContentPart.java × | MainVerticle.java ×

```
@Override
public void start(Future<Void> startFuture) throws Exception {

    wikiDbQueue = config().getString(CONFIG_WIKIDB_QUEUE, "wikidb.queue");
    dbService = WikiDatabaseService.createProxy(vertx, wikiDbQueue);

    HttpServer server = vertx.createHttpServer(new HttpServerOptions()
            .setSsl(true)
            .setKeyStoreOptions(new JksOptions()
                    .setPath("server-keystore.jks")
                    .setPassword("secret")
            )
    );

    AuthProvider auth = ShiroAuth.create(vertx, new ShiroAuthOptions()
            .setType(ShiroAuthRealmType.PROPERTIES)
            .setConfig(new JsonObject()
                    .put("properties_path", "classpath:wiki-users.properties")
            )
    );

    webClient = WebClient.create(vertx, new WebClientOptions()
            .setSsl(true)
            .setUserAgent("vert-x3"));

    Router router = Router.router(vertx);
    router.route().handler(CookieHandler.create());
    router.route().handler(BodyHandler.create());
    router.route().handler(SessionHandler.create(LocalSessionStore.create(vertx)));
    router.route().handler(UserSessionHandler.create(auth));

    AuthHandler authHandler = RedirectAuthHandler.create(auth, "/login");
    router.route("/").handler(authHandler);
    router.route("/wiki/*").handler(authHandler);
    router.route("/action/*").handler(authHandler);
```

GOING REACTIVE

With Vert.x integration with React

INTEGRATION WITH RXJAVA

- Done via the dependency: vertx-rx-java2

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-rx-java2</artifactId>
</dependency>
```

To create a Verticle we do not extend *io.vertx.core.AbstractVerticle* anymore, but:

io.vertx.reactivex.core.Vertx

Thus we have an idiomatic API extension

WHY USING RXJAVA?

- The callback based API of Vert.x stack works well but it can become a bit tedious when we have to handle several source of events and deal with complex data flows.

- That‘s where RxJava shines and Vert.x integrates well with it.

EXAMPLE: THE MAINVERTICLE

MainVerticle.java

```
27   */
28  public class MainVerticle extends AbstractVerticle {
29
30    @Override
31    public void start(Future<Void> startFuture) throws Exception {
32
33      // tag::rx-deploy-verticle[]
34      Single<String> dbVerticleDeployment = vertx.rxDeployVerticle(
35        "io.vertx.guides.wiki.database.WikiDatabaseVerticle");
36      // end::rx-deploy-verticle[]
37
38      // tag::rx-sequential-composition[]
39      dbVerticleDeployment
40        .flatMap(id -> { // <1>
41
42          Single<String> httpVerticleDeployment = vertx.rxDeployVerticle(
43            "io.vertx.guides.wiki.http.HttpServerVerticle",
44            new DeploymentOptions().setInstances(2));
45
46          return httpVerticleDeployment;
47        })
48        .subscribe(id -> startFuture.complete(), startFuture::fail); // <2>
49      // end::rx-sequential-composition[]
50    }
51  }
52
```

CHANGES IN DEPLOYMENT APPROACH

- The rxDeploy method does not take a Handler<AsyncResult<String>>
- As a final Parameter, instead it returns a Single<String>

- Besides, the operation does not start when the method is called.
- *It starts when you subscribe to the Single*. When the operation completes,
- it emits the deployment id or signals the cause of the problem with a Throwable.

EXECUTING AUTHORIZATION QUERIES CONCURRENTLY

- In the previous example, we saw how to use *RxJava operators* and the Rxified Vert.x API to *execute asynchronous operations in order*.
- But *sometimes* this guarantee is not required, or *you simply want them to run concurrently for performance reasons.*

- The JWT token generation process in the HttpServerVerticle is a good example of such a situation.
- *To create a token, we need all authorization queries to complete, but queries are independent from each other*:

```
auth.rxAuthenticate(creds).flatMap(user -> {
  Single<Boolean> create = user.rxIsAuthorised("create"); (1)
  Single<Boolean> delete = user.rxIsAuthorised("delete");
  Single<Boolean> update = user.rxIsAuthorised("update");

  return Single.zip(create, delete, update, (canCreate, canDelete, canUpdate) -> {
    return jwtAuth.generateToken(
      new JsonObject()
        .put("username", context.request().getHeader("login"))
        .put("canCreate", canCreate)
        .put("canDelete", canDelete)
        .put("canUpdate", canUpdate),
      new JWTOptions()
        .setSubject("Wiki API")
        .setIssuer("Vert.x"));
  });
}).subscribe(token -> {
  context.response().putHeader("Content-Type", "text/plain").end(token);
}, t -> context.fail(401));
```

Bridging the gap between callbacks and RxJava

At times, you may have to mix your RxJava code with a callback-based API. For example, service proxy interfaces can only be defined with callbacks, but the implementation uses the Vert.x Rxified API.

In this case, the `io.vertx.reactivex.SingleHelper.toObserver` class can adapt a `Handler<AsyncResult<T>>` to an RxJava `SingleObserver<T>`:

```
@Override
public WikiDatabaseService fetchAllPagesData(Handler<AsyncResult<List<JsonObject>>
  dbClient.rxQuery(sqlQueries.get(SqlQuery.ALL_PAGES_DATA))
    .map(ResultSet::getRows)
    .subscribe(SingleHelper.toObserver(resultHandler));   (2)
  return this;
}
```

DEMO

WIKIPEAKS as a SPA