WBA: Microservice Developer Guide

Microservice Developer Guide - v 3.1.0

REVISION HISTORY

Revision #	Author	Date	Description	Reviewed By
1.0.0	Lohes	23-March-2018	Initial Draft	Jeb
1.1.0	Lohes	27-March-2018	Doc title, diagram and section alignment changes	
1.2.0	Lohes	30-March-2018	Config map changes	
1.3.0	Lohes	02-April-2018	Internal connectivity within the applications	
1.4.0	Lohes	13-April-2018	Secrets management in K8S	
1.5.0	Lohes	17-April-2018	Jenkins master - slave configuration	
1.6.0	Lohes	20-April-2018	Update in technology stack and secrets management	
1.7.0	Lohes	17-May-2018	Tools update	
1.8.0	Lohes	30-May-2018	Pact integration	
1.9.0	Lohes	08-June-2018	Sitespeed integration	
2.0.0	Lohes	11-June-2018	Artillery integration	
2.1.0	Lohes	18-June-2018	Service Virtualization with Mountebank	
3.0.0	Lohes	21-June-2018	Merged Spring Boot developer guide	
3.0.1	Lohes	08-August-2018	Minor updates to npm scripts	

3.1.0	Lohes	10-Sep-2018	Virtual service CI/CD pipeline	

PURPOSE OF THIS DOCUMENT

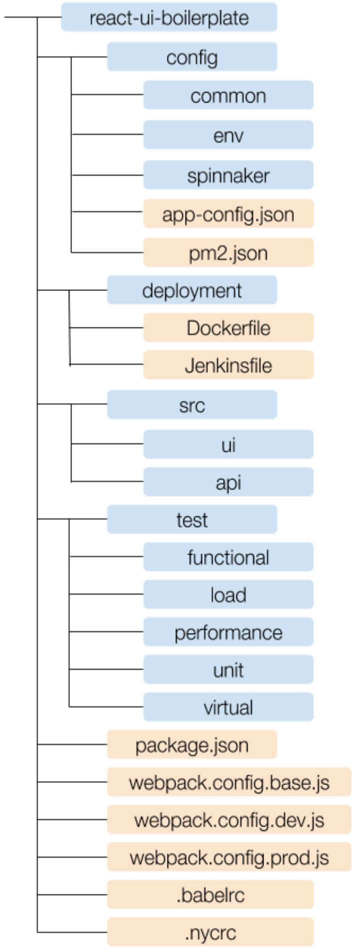
The objective of this document is to explain the key principles for designing and implementing high performing microservices with continuous deployment/delivery model.

This document will serve the following purposes,

- Discusses technology choices across layers of the application and guides the decision making
- Provides development, deployment and operational guidelines

REACT UI BOILERPLATE

Application Folder Structure

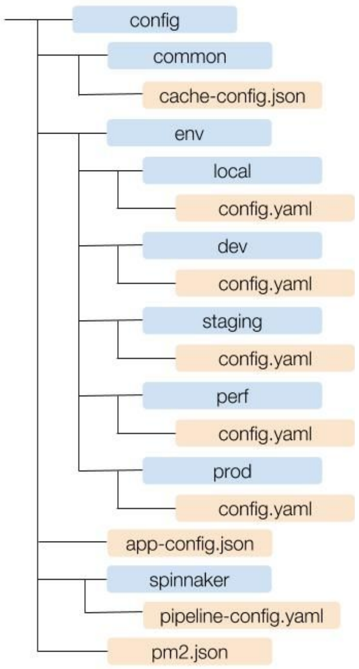


react-ui-boilerplate: This is the root directory of the page. Under this directory, we will have the below

config	● **common:** Contains the configurations of node layer caching ● **env:** Contains the environment specific application configurations ● **spinnaker:** It has the configurations which is used to create pipeline in Spinnaker ● **app-config.json:** It has the application configurations that are common for all the environments ● **pm2.json:** It has the PM2 module configurations which is used to start the application at higher environments
deployment	● **Dockerfile:** Dockerfile to build the docker image ● **Jenkinsfile:** Jenkins file with the scripts for the different stages of the pipeline. Jenkins job will refer this file to create the pipeline
src	This directory contains the source code of reactjs/nodejs microservice
test	● **unit:** This directory contains the UI/API test spec files ● **functional:** This directory contains the browser specific config files, feature files and step definition files ● **performance:** It has the WebPageTest configurations like application URL, SLA for css/js size and count for different breakpoints. ● **load:** It has JMeter JMX file with configurations like loops, no of threads, ramp-up period ● **virtual:** It contains files like imposters, stubs

package.json	It consists of ● **dependencies:** Contains the modules that are required at the runtime ● **devDependencies:** Contains the dependencies that are needed at the dev environment ● **scripts:** Application scripts like like build, start, unit test are defined here
webpack.config	● **webpack.config.base.js:** It has the webpack configurations that are common for local dev and higher environments ● **webpack.config.dev.js:** It has the webpack configurations that creates the bundle file with debug states and other features that are must for dev environment ● **webpack.config.prod.js:** It creates the production ready bundle files with optimizations, versioning and etc.,
.babelrc	Babel configurations to enable the transforms during build, test and etc
.nycrc	It has the Istanbul configurations like transpiler plugin, report type, report directory, threshold definition for code coverage and etc.,

Application Config Folder Structure



- **config/common/cache-config.json:** This file has the configurations like cache key, ttl and disk cache path.

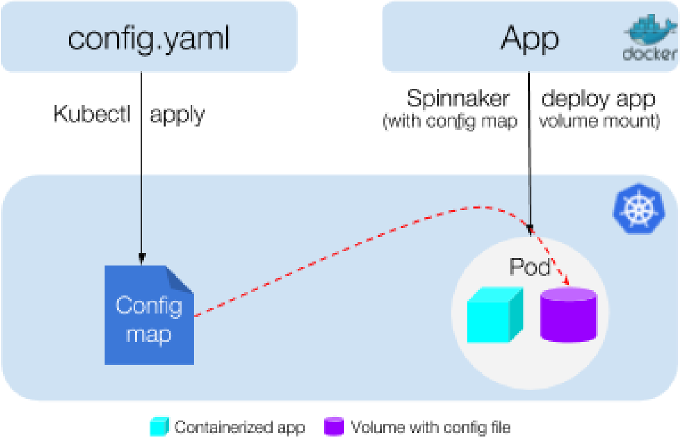
```
{
    "dataCache": {
        "header": {
            "key": "d_header",
            "ttl": 172800,
            "enableCache":true
        }
    },
    "componentCache":{
        "header": {
            "key": "c_header",
            "ttl": 172800,
            "enableCache":true
        }
    },
    "cachePath":{
        "local_path" : "diskCache",
        "env_path" : "/usr/local/ecomm/data/vpd"
    }
}
```

- **config/app-config.json:** It has app configs that are common for all the environments and configs that changes per environment are parameterised. In the below example, values of `env_atg.client.url`, `env_atg.server.url`, `env_atg.server.proxy`, `env_functional_test_url` and `env_log_config_location` are parameterized since it changes per environment.

```
{
    "env_app_name": "boilerplate",
    "env_config_server_port": 8080,
    "env_app_context": "reactui",
    "env_atg": {
        "client": {
            "url": "${ENV_CSR_ATG_SERVICE_URL}",
            "proxy": false
        },
        "server": {
            "url": "${ENV_SSR_ATG_SERVICE_URL}",
            "proxy": "${ENV_SSR_ATG_SERVICE_ENABLE_PROXY}"
        }
    },
    "env_headerui": {
        "url": "/common/v1/headerui",
        "timeout": {
            "server": 1000,
            "client": 1000
        }
    },
    "env_server_error_link": "/servererror.jsp",
    "env_functional_test_url": "${ENV_FUNCTIONAL_TEST_URL}",
    "env_log_config_location": "${ENV_LOG_CONFIG_LOCATION}",
    "env_log_level": "INFO",
    "env_log_accesslog_rotation_policy": "5MB",
    "env_log_applicationlog_rotation_policy": "5MB",
    "env_log_accesslog_location": "",
    "env_log_applicationlog_location": ""
}
```

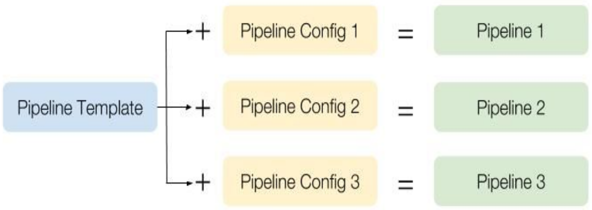
- **config/env/<env>/config.yaml:** It has the properties that changes per environment and takes the Kubernetes config-map structure.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: reactui-boilerplate-config
  namespace: dotcom-test-dev
data:
  config.yaml: |
    ENV_CSR_ATG_SERVICE_URL: "https://m-int1.walgreens.com"
    ENV_SSR_ATG_SERVICE_URL: "http://m-int1.walgreens.com"
    ENV_SSR_ATG_SERVICE_ENABLE_PROXY: "true"
    ENV_LOG_CONFIG_LOCATION: "LOCAL"
    ENV_FUNCTIONAL_TEST_URL: "https://m-int1.walgreens.com"
```



  - **ENV_FUNCTIONAL_TEST_URL** and **ENV_LOG_CONFIG_LOCATION** are the two variables defined in config/app-config.json. Environment specific config.yaml files will have the values of these keys(ENV_LOG_CONFIG_LOCATION and ENV_FUNCTIONAL_TEST_URL) as shown above.
  - In local development environment, this config files will be read and then merged with config/app-config.json file.
  - In higher environment(dev, staging, perf and prod), config-map(with reactui-boilerplate-config name) will be created in Kubernetes using kubectl. This config-map will be mounted as volume to the pod and application will read this file then replaces the variables of app-config.json with values of the file.
  - If the variables defined in app-config.json is not found in config.yaml, then it will look for these keys in environment variables.

- **config/spinnaker/pipeline-config.yaml:** Pipeline template with the parameters will be published to the Spinnaker. This pipeline-config.yaml will have the values of the variables defined in pipeline template and then it will be used to create the Spinnaker pipeline.



- **config/pm2.json:** PM2 is the general purpose process manager with simple and efficient process management. The config has the node ENV variable, application startup script.

```
{
    "apps" : [
        {
            "name":"react-ui-boilerplate",
            "script":"dist/server-bundle.js",
            "env_production" : {
                "NODE_ENV": "production"
            }
        }
    ]
}
```

Application Deployment

- **react-ui-boilerplate/deployment/Dockerfile:** It is a text file that contains all the commands to assemble an image. Docker reads the instructions from this file and builds the image. Application startup command will be part of the spinnaker config file.

```
FROM nonprodregistry.azurecr.io/alpine-node:6
ENV HOME_PATH=/usr/react-ui-boilerplate

RUN mkdir -p ${HOME_PATH}
WORKDIR ${HOME_PATH}
COPY /build/react-ui-boilerplate.zip ${HOME_PATH}

RUN unzip  ${HOME_PATH}/react-ui-boilerplate.zip -d ${HOME_PATH}
RUN rm -f ${HOME_PATH}/react-ui-boilerplate.zip
```

- **react-ui-boilerplate/deployment/Jenkinsfile:** It is a text file that contains the definition of a Jenkins Pipeline. Jenkins job will refer this file to create stages of the pipeline.

```
  node {
     stage('Clone') {
       ...
     }
     stage('CSS Lint') {
       ...
     }
     stage('JS Lint') {
        ...
     }
     stage('Unit Test') {
       ...
     }
     stage('Build') {
       ...
     }
}
```

Application Package

- **react-ui-boilerplate/package.json:** Three major fields in package.json are
    - **devDependencies**: Has modules which are only required during development(static code analysis, unit test, build, functional test and etc.,)
    - **dependencies**: Contains the modules which are only required at runtime
    - **scripts**: Contains script commands that are run at various times in the life cycle. The key is the lifecycle event, and the value is the command to run at that point.

  Note: Node module version should must match exact version. Modules should not auto upgrade when there is any new version available.

```
{
    "name": "wag-ui-boilerplate",
    "version": "1.0.0",
    "description": "Walgreens UI microservices boilerplate template",
    "main": "index.js",
    "scripts": {
        "build-prod": "npm run install-common && npm install && npm run webpack",
        "start": "NODE_ENV=production node dist/server-bundle.js"
        ...
    },
    "author": "",
    "license": "",
    "devDependencies": {
        "webpack": "2.2.1"
         ...
    },
    "dependencies": {
        "express": "4.15.2",
         ...
    }
}
```

NPM Scripts

npm run unit-test	Executes the Mocha unit test scripts and produces HTML report in ./test/unit/reports/status/index.html and ./test/unit/reports/coverage/index.html
npm run pact-publish	Publish the pact file(created during unit test) to pact broker
npm run pact-verify-provider	Run the pact script to verify the contracts published by consumers

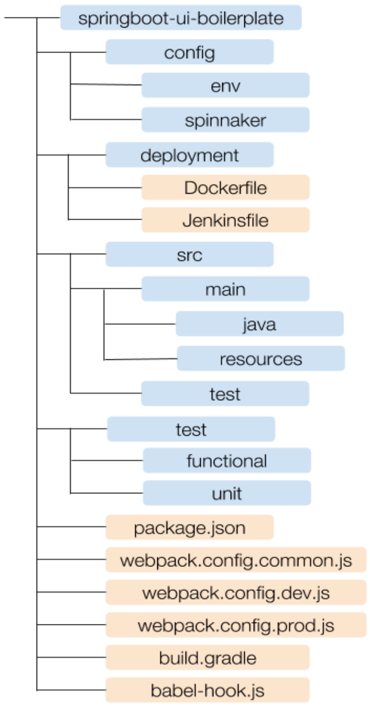
npm run sonarqube-scanner	Executes static code analysis and code coverage in the given sonarqube
npm run uifunctional-test	Executes the BDD functional test scripts and generates the HTML report at ./test/functional/reports/index.html
npm run ui-perf-test-dev	Executes the UI performance test(Dev environment) using sitespeed
npm run ui-perf-test-qa	Executes the UI performance test(QA environment) using sitespeed
npm run load-test	Executes the load test using artillery
npm run load-test-report	Converts the json report generated by artillery to HTML report
npm run build-dev	Does the webpack build with watcher which listens for changes in the source files and creates the build with the latest changes
npm run build-prod	Creates the production ready build with all the optimizations like minification, versioning and etc..
npm run start-dev	Starts the node server with local configurations
npm run pm2-start	Starts the node server using PM2 with node env variable as production
npm run start	Starts the node server with node env variable as production
npm run pack	Creates a zip file with dist files and runtime dependencies

Technology Stack

Checkout	Code Analysis	Unit Test	Build	Deploy to Dev	Functional Test	Load Test	UI Perf Test
Checks out the specified Git branch in Jenkins slave agent	Performs static code analysis for CSS and JS in Jenkins slave agent	Runs unit tests and enforces code coverage in Jenkins slave agent	Packages the application and creates the Docker image in Jenkins slave agent	Deploys the docker image into the Kubernetes ACS cluster	Triggers UI functional test in Jenkins slave agent	Triggers Load test in Jenkins slave agent	Triggers UI performance test in Jenkins slave agent
Stash	sonarqube	airbnb Enzyme istanbul chai NOCK cucumber	npm webpack docker	kubernetes Spinnaker	Protractor chai cucumber	mountebank ARTILLERY.IO	sitespeed.io

SPRING BOOT UI BOILERPLATE

Application Folder Structure

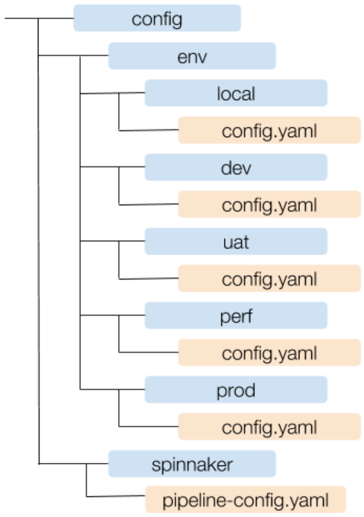


springboot-ui-boilerplate: This is the root directory of the page. Under this directory, we will have the below

config	● **env:** Contains the environment specific application related properties. ● **spinnaker:** contains configuration related to spinnaker. This will be maintained by Devops team and Dev team won't have access to edit this file.
deployment	● **Jenkinsfile:** Jenkins file with the scripts for the different stages of the pipeline. Jenkins job will refer this file to create the pipeline
src	● **main/java:** This directory contains java/thymeleaf/js/css source code and java test cases (unit test case and pact) ● **main/resources:** This directory contains UI components ● **test:** This directory contains java unit test cases
test	● **unit:** This directory contains the UI test spec files ● **functional:** This directory contains the browser specific config files, feature files and step definition files
package.json	It consists of ● **dependencies:** Contains the modules that are required at the runtime ● **devDependencies:** Contains the dependencies that are needed at the dev environment ● **scripts:** Application scripts like like build, start, static are defined here

webpack.config	● **webpack.config.common.js:** It has the webpack configurations that are common for local dev and higher environments ● **webpack.config.dev.js:** It has the webpack configurations that creates the bundle file with debug states and other features that are must for dev environment ● **webpack.config.prod.js:** It creates the production ready bundle files with optimizations, versioning and etc.,
build.gradle	Gradle configuration file. It consists of ● Maven Repository Configuration ● JAR Dependencies ● Build Tasks

Application Config Folder Structure



- All the application related properties can be maintained in application.yaml and properties(Eg:logpath, log pattern) which need to be initialized before server startup should be maintained in bootstrap.yaml
- The environment related properties should be maintained in config.yaml under environment specific folders in config/env path. The config.yaml will be in the below format and it takes the kubernetes configMap structure.
- In the config.yaml file all the environment related properties should be under config.yaml and the bootstrap properties which need to be overridden at env layer should be under bootstrap.yaml

application.yml

```
spring:
    thymeleaf:
        cache: false
        check-template-location: true
        encoding: UTF-8
        prefix: classpath:/dist/templates/
        servlet:
            content-type: text/html
        suffix: .html
mongodb:
  mongoCertificateFilePath: ${user.dir}/src/main/resources/mongoclient.jks
```

bootstrap.yml

```
log:
    history: 10
    maxLogFileSize: 100MB
    pattern: "%d [${spring.application.name},%X{X-B3-TraceId},%X{X-B3-SpanId}] ${PID:- } [%thread] %-5level %logger{35} - %msg %n"
spring:
    application:
        name: springboot-ui-boilerplate
server:
  tomcat:
    basedir: /Users/sv/Documents/boilerplatelogs/
```

/config/env/*/config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: springbootuiboilerplate-config
```

```
data:
  config.yaml: |
    mongodb:
      mongoCertificateFilePath: /deployments/secrets/mongoclient.jks
    voltage:
          policyURL: 'https://voltage-pp-0000.devtest.pmt.walgreens.com/policy/clientPolicy.xml'
          sharedSecret: voltage123
          trustStorePath: /usr/local/ecomm/voltage-dependencies/trustStore
          nativeLibraryFilePath: /usr/local/ecomm/voltage-dependencies/linux/
    ## Example for calling other services. Configuring Host and Port
    ## rxStatusHostName: ${DPFRXSTATUS_SERVICE_HOST}
    ## rxStatusPort: ${DPFRXSTATUS_SERVICE_PORT}
  bootstrap.yaml: |
    log:
        history: 10
        maxLogFileSize: 100MB
    server:
      tomcat:
        basedir: /deployments/logs/springbootuiboilerplate/${HOSTNAME}
```

Gradle Configuration

Below are the scripts in Build.gradle file

- BuildScripts
  - Spring Boot Gradle Plugin – To Build the Spring Boot 2 Application
  - Gradle Docker Plugin – To create the DockerFile at build time
  - Gradle Sonarqube – Static Code Analysis at Build Time
  - Pact JVM Gradle – Build Time Dependency to generate the pact file/ verify the pact
  - Gradle Node Plugin – To run npm commands(install, run build,test,etc.,) from Gradle
  - Gradle Git Properties – To show the Git Properties at /info endpoint

```
buildscript {
    ext {
        springBootVersion = '2.0.1.RELEASE'
    }
    repositories {
        mavenCentral()
        maven {
            url = 'http://wagwiki.walgreens.com/artifactory/ecomm-snapshot-libs'
            credentials {
            }
        }
        maven{
            url='https://plugins.gradle.org/m2/'
        }
        mavenLocal()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
        classpath ("com.bmuschko:gradle-docker-plugin:3.2.1")
        classpath ("org.sonarsource.scanner.gradle:sonarqube-gradle-plugin:2.5")
        classpath("au.com.dius:pact-jvm-provider-gradle_2.12:3.5.13")
        classpath ("com.moowork.gradle:gradle-node-plugin:1.2.0")
    }
}
plugins {
    id "com.gorylenko.gradle-git-properties" version "1.4.17"
}
```

- Build Script Repositories
    - Maven Central
    - Artifiactory – Walgreens Enterprise Maven Repository Manager – http://wagwiki.walgreens.com/artifactory/ecomm-snapshot-lib. All the custom JARs(IC+,loyalty, OWG,etc.,) and Starter Projects should be pushed to JFrog Artifactory
    - Gradle Plugins Repository - https://plugins.gradle.org/m2/ (Gradle Node Plugin)
    - MavenLocal - .m2 Folder of the local systems or build server.
- Build Dependencies

- Spring Boot Version: 2.0.0 RELEASE
- Spring Cloud Version: Finchley.M9 (Will be upgraded to Finchley Stable Version before application goes live and Finchley Release is planned on Apr 18th 2018).
  - Spring-Boot-Starter-Web: Rest,Tomcat,Logging,AutoConfiguration
    ```
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-web'
    ```
  - Spring-Boot-Starter-data-Mongodb: MongoDB
    ```
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-mongodb'
    ```
  - Spring-Boot-Starter-data-redis: Redis
    ```
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-redis'
    ```
  - Spring-Boot-Starter-actuator: Monitoring
    ```
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-actuator'
    ```
  - Spring-Cloud-Starter-Sleuth: Tracing
    ```
    compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-sleuth'
    ```
  - Spring-cloud-starter-netflix-hystrix: Hystrix (Circuit Breaker)
    ```
    compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-hystrix'
    ```
  - Spring-cloud-starter-netflix-hystrix: Hystrix Dashboard
    ```
    compile group: 'org.springframework.cloud', name: 'spring-cloud-starter-hystrix-dashboard'
    ```
  - Spring-Boot-Starter-Thymeleaf: Thymeleaf
    ```
    compile group: 'org.springframework.cloud', name: 'spring-boot-starter-thymeleaf'
    ```
  - SpringFox-Swagger: Swagger API
    ```
    compile group: 'io.springfox', name:'springfox-swagger2'
    ```
  - SpringFox-Swagger: Swagger UI
    ```
    compile group: 'io.springfox', name:'springfox-swagger-ui'
    ```
  - Spring-Boot-Starter-Voltage: Voltage(Walgreens)
    ```
    compile group: 'com.walgreens', name:'spring-boot-starter-voltage'
    ```
  - Spring-Boot-Starter-jwt: JWT (Walgreens)
    ```
    compile group: 'com.walgreens', name:'spring-boot-starter-jwt'
    ```
  - Spring-Boot-Starter-devicedetection: Device Detection (Walgreens)
    ```
    compile group: 'com.walgreens', name:'spring-boot-starter-devicedetection'
    ```
  - Lombok – To Generate Getters, Setters, ToString during compile time
    ```
    compileOnly group: 'org.projectlombok', name: 'lombok'
    ```
  - Spring Boot Hot Reload

```
compile group: 'org.springframework.boot', name:'spring-boot-devtools'
```

- Test Dependencies
    - Spring-Boot-Starter-Test: Unit Test Cases

      ```
      testCompile group: 'org.springframework.boot', name: 'spring-boot-starter-test'
      ```
    - Spring-WS-Test: Spring Framework Test Cases

      ```
      testCompile group: 'org.springframework.ws', name: 'spring-ws-test', version: '2.4.0.RELEASE'
      ```
    - Embedded Mongo

      ```
      testCompile group: 'de.flapdoodle.embed', name: 'de.flapdoodle.embed.mongo', version: '2.0.1'
      ```
    - Embedded Redis

      ```
      testCompile group: 'com.github.kstyrc', name: 'embedded-redis', version: '0.6'
      ```
    - Pact

      ```
      testCompile group: 'au.com.dius', name: 'pact-jvm-provider-gradle_2.12', version: '3.5.7'
      testCompile group: 'au.com.dius', name: 'pact-jvm-provider-junit_2.12', version: '3.5.13'
      testCompile group: 'au.com.dius', name: 'pact-jvm-consumer-junit_2.12', version: '3.5.13'
      ```
- Sonarqube - Static code analysis.
    - Below is the SonarQube configuration and the respective gradle task command.
    - Gradle script: `gradle sonarqube -Dsonar.host.url=http://172.20.7.133:8090/sonarqube`

      ```
      sonarqube {⍰
          properties {⍰
              property 'sonar.projectName', 'SpringBoot BoilerPlateUI for Gradle Usage'⍰
          }
       }
      ```
- Junit Test Code Coverage Report – Jacaco Plugin
    - Gradle script to run the Unit Test: `gradle test`

      ```
      test {
          ignoreFailures = true
          reports.junitXml.enabled = false
          reports.html.enabled = true
          testLogging {
              exceptionFormat = 'full'
          }
          jacoco {
              append = false
              destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
      ```

```
        maxHeapSize = '2048m'
        minHeapSize = '1024m'
        //classDumpDir = file("$buildDir/jacoco/classpathdumps")
    }
}
jacocoTestReport {
    group = "Reporting"
    reports {
        xml.enabled true
        csv.enabled false
        html.destination "${buildDir}/reports/coverage"
    }
    afterEvaluate {
        classDirectories = files(classDirectories.files.collect {
            fileTree(dir: it,
                    exclude: [
                    ])
        })
    }
}
test.finalizedBy jacocoTestReport
jacoco {
    toolVersion = "0.7.6.201602180812"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

- Generation of DockerFile
    - Gradle script to create the DockerFile: `gradle createDockerfile`

```
task createDockerfile(type: Dockerfile) {
    println 'createDockerfile'
    //Dockerfile path
    destFile = project.file('Dockerfile')
    //Base Image
    from 'nonprodregistry.azurecr.io/baseimg_gradle_microservices:v1'
    copyFile '/build/voltage-dependencies/', '/usr/local/ecomm/voltage-dependencies/'
    //Labels
    def labels = [
        'com.walgreens.container.image.license': 'Walgreens Co',
        'com.walgreens.container.image.name'   : '' + project.name
    ]
    label labels
    //Add Springboot Jar
    addFile '/build/libs/' + project.name + '.jar', '/usr/local/ecomm/app/' + project.name + '.jar'
    //Maintainer
    maintainer 'Walgreens Co'
}
```

- Pact - Generation of Pact File and publish the file to pact broker (Consumers/Clients)
    - Gradle script: `gradle pactPublish`

```
pact {
    serviceProviders {
        rxorder {
            publish {
                pactDirectory = '/Users/sv/Documents/springboot-ui-boilerplate/target/pacts'//defaults to
$buildDir/pacts
                pactBrokerUrl = 'http://localhost:80'
                version=2.0
            }
        }
    }
}
```

- Pact Verify (Producers/Server)
    - Gradle Command: `gradle pactVerify`

```
pact {
    serviceProviders {
        profile {
            startProviderTask = 'startProvider'
            terminateProviderTask = 'stopProvider'
            //protocol ='http'
            host = 'localhost'
            port = 8090
            //hasPactsFromPactBroker('http://localhost:80/')
        }
```

```
            }
        }
        task startProvider(type: SpawnProcessTask, dependsOn: 'assemble') {
            command "java -jar ${jar.archivePath}"
            ready 'Started MainApplication'
        }
        task stopProvider(type: KillProcessTask) {
            print("Hello stop");
        }
```

NodeJS Configuration

- **springboot-ui-boilerplate/package.json:** Three major fields in package.json are
    - **devDependencies**: Has modules which are only required during development(static code analysis, unit test, build, functional test and etc.,)
    - **dependencies:** Contains the modules which are only required at runtime
    - **scripts**: Contains script commands that are run at various times in the life cycle. The key is the lifecycle event, and the value is the command to run at that point.

  Note: Node module version should must match exact version. Modules should not auto upgrade when there is any new version available

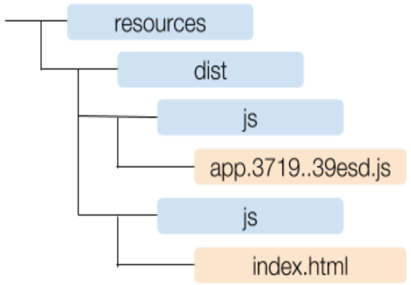
```
{
    "name": "SpringBoot-Javascript-Boilerplate",
    "version": "1.0.1",
    "author": "",
    "description": "",
    "main": "src/index.js",
    "scripts": {
        "sonar-scanner": "sonar-scanner -Dsonar.host.url=http://172.20.7.133:8090/sonarqube
-Dsonar.projectKey=springboot-ui-boilerplate -Dsonar.projectName=springbootuiboilerplate -Dsonar.sources=src -D
sonar.tests=test/unit -Dsonar.javascript.lcov.reportPaths=coverage/lcov.info",
        "start": "NODE_ENV=development webpack-dev-server --config webpack.config.dev.js",
        "start:dev": "webpack-dev-server --open",
        "build": "webpack -p --config webpack.config.prod.js",
        "unit-test": "cross-env NODE_ENV=test node node_modules/nyc/bin/nyc.js --reporter=lcov mocha --require
babel-register ./test/**/*.spec.js --require jsdom-global --require mock-local-storage --require
jsdom-global/register",
        "watch": "webpack --config webpack.config.dev.js --watch",
        "e2e-test": "npm install -g protractor && webdriver-manager update && protractor 
test/functional/chrome.config.js"
    },
    "devDependencies": {
        "chai": "4.1.2",
        "cucumber": "1.3.1",
        "mocha": "5.0.4",
        "nyc": "10.2.0",
        "webpack": "3.5.5"
    },
    "dependencies": {

    }
}
```

NPM Scripts

npm run unit-test	Executes the mocha unit test scripts and produces HTML report
npm run sonar-scanner	Executes static code analysis and code coverage in the given sonarqube
npm run e2e-test	Executes the BDD functional test scripts and generates the HTML report
npm run build	Creates the production ready build with all the optimizations like minification, versioning and etc..
npm run watch	Does the webpack build with watcher which listens for changes in the source files and creates the build with the latest changes
npm run start	Starts the node server

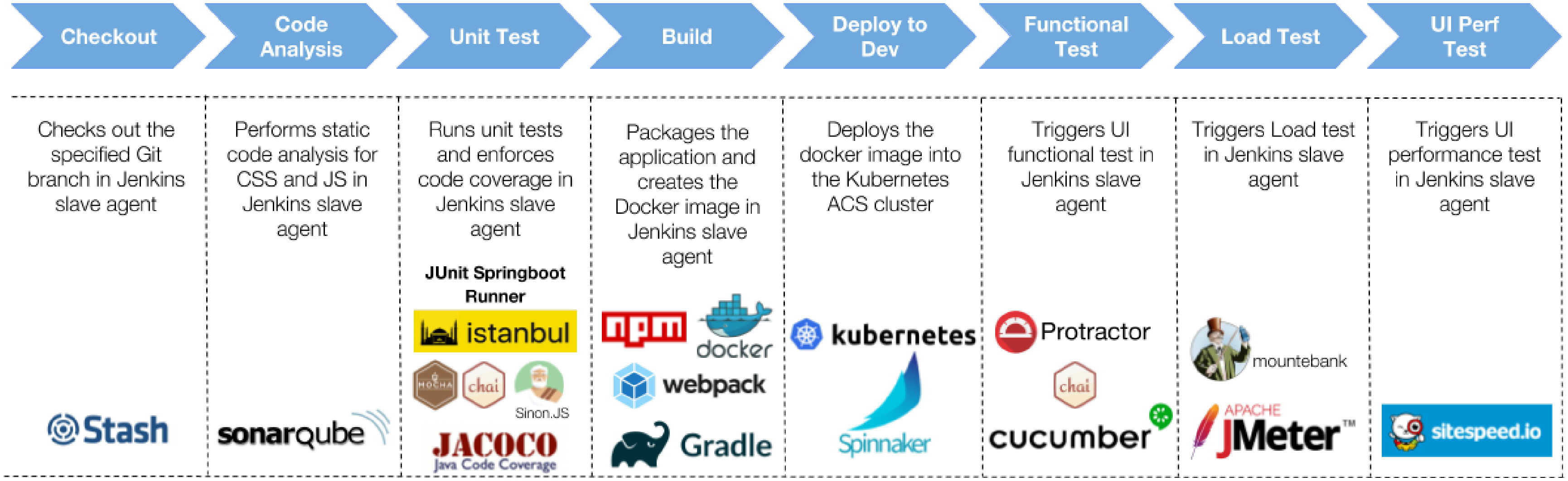
Versioning and Minification



resources
dist
js
app.3719..39esd.js
js
index.html

- Webpack minifies the JS and CSS and create versioned files in dist folder and rename the references in Thymeleaf template.
- Thymeleaf refers the static folder for JS and CSS and HTML file for templates by default. This default behavior is overridden in WebConfig.java to refer to dist folder.

Technology Stack



Checkout	Code Analysis	Unit Test	Build	Deploy to Dev	Functional Test	Load Test	UI Perf Test
Checks out the specified Git branch in Jenkins slave agent	Performs static code analysis for CSS and JS in Jenkins slave agent	Runs unit tests and enforces code coverage in Jenkins slave agent	Packages the application and creates the Docker image in Jenkins slave agent	Deploys the docker image into the Kubernetes ACS cluster	Triggers UI functional test in Jenkins slave agent	Triggers Load test in Jenkins slave agent	Triggers UI performance test in Jenkins slave agent
Stash	sonarqube	JUnit Springboot Runner istanbul MOCHA chai Sinon.JS JACOCO Java Code Coverage	npm docker webpack Gradle	kubernetes Spinnaker	Protractor chai cucumber	mountebank APACHE JMeter™	sitespeed.io

CUCUMBER UNIT TESTING

Cucumber is a tool for running automated tests written in plain language. Because they're written in plain language, they can be read by anyone in the team.

Feature file :

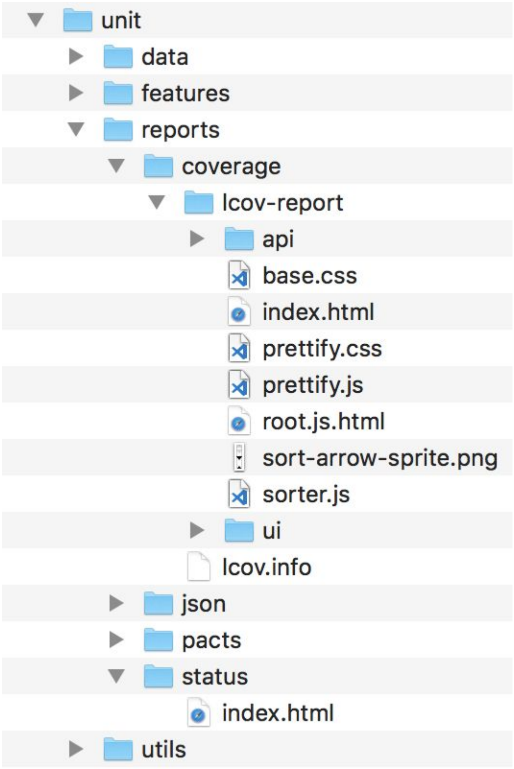
```
Feature: Root component
 Root component renders the page specific components
@rootComponent
 Scenario: Render root component
   Given Root component is available
   Then Welcome to boilerplate text shown
```

Step definition :

```
Given('Root component is available', function(callback) {
    let storeData = readJsonFile('../../../data/initial-data.json');
    store = mockStore(storeData);
    const renderComp = mount(
        <Provider store={store}>
            <Root />
        </Provider>
    );
    this.setTo(renderComp);
    callback();
});
Then('Welcome to boilerplate text shown', function(callback) {
    expect(this.response.find('.root-page').text()).to.equal('Welcome to Boilerplate');
    callback();
});
```

NPM Script :

```
"unit-test": "NODE_ENV=test ENV_SERVER=local node --max_old_space_size=4096 node_modules/nyc/bin/nyc.js
./node_modules/.bin/cucumber-js test/unit/features/ui/*.feature --format
json:test/unit/reports/json/results.json ; node test/unit/utils/json-to-html.js"
```



```
▼ unit
    ▶ data
    ▶ features
    ▼ reports
        ▼ coverage
            ▼ lcov-report
                ▶ api
                  base.css
                  index.html
                  prettify.css
                  prettify.js
                  root.js.html
                  sort-arrow-sprite.png
                  sorter.js
                ▶ ui
              lcov.info
        ▶ json
        ▶ pacts
        ▼ status
              index.html
    ▶ utils
```

- Istanbul(nyc) will create lcov report which is used by sonarqube for coverage details.
- Cucumber js will create test case status report as json which will then be converted into HTML report.

CDC TESTING USING PACT

- Consumer Driven Contracts is a pattern that drives the development of the Provider from its Consumers point of view.
- Pact is a testing tool that guarantees those Contracts are satisfied.
- Consumer - Creates pact file as part of unit test, publishes the pact file to the pact broker. The Pact Broker is an application for sharing for consumer driven contracts(pact files) and verification results.
- Provider gets pact files from pact broker and validates.

Consumer

- As part of unit testing, Consumer will mock all APIs that are consumed using the mock server provided by pact. Pact provider API will bring up a mock server by default. The API endpoints that needs to be mocked has to added to the provider using `provider.addInteraction({...}).` All these steps needs to be done in hooks like `Before` or `BeforeAll.`

```
import { Given, Then, AfterAll, Before } from 'cucumber';
const { Pact } = require('@pact-foundation/pact');
const MOCK_SERVER_PORT = 8080;
const provider = new Pact({
    consumer: 'reactuiboilerplate',
    provider: 'reactuiboilerplate',
    host: 'localhost',
    port: MOCK_SERVER_PORT,
    ssl: true,
    sslcert: '../wag-common-ui/config/env/local/cert/walgreens.com.crt',
    sslkey: '../wag-common-ui/config/env/local/cert/walgreens.com.key',
    log: path.resolve(process.cwd(), 'logs', 'pact.log'),
    dir: path.resolve(process.cwd(), 'test/unit/reports/pacts'),
    logLevel: 'INFO',
    spec: 2
});

const EXPECTED_BODY = {'status': 'UP', 'appName': 'reactuiboilerplate'};
Before(function(testCase, callback) {
    provider.setup()
    .then(() => {
        provider.addInteraction({
            state: 'V1 health check data',
            uponReceiving: 'a request for V1 health check data',
            withRequest: {
                method: 'GET',
                path: '/reactuiboilerplate/v1/health',
            },
            willRespondWith: {
                status: 200,
                body: EXPECTED_BODY
            }
        });
    })
    .then(() => callback());
});
```

- Once all the test cases are executed(`AfterAll` hook), `provider.finalize()` will be invoked which will create the pact file in the given directory.

```
AfterAll(function () {
    provider.finalize();
});
```

Below is the sample pact file.

```
{
    "consumer":{
        "name":"reactuiboilerplate"
    },
    "provider":{
        "name":"reactuiboilerplate"
    },
    "interactions":[
        {
            "description":"a request for V1 health check data",
            "providerState":"V1 health check data",
            "request":{
                "method":"GET",
                "path":"/reactuiboilerplate/v1/health"
            },
            "response":{
                "status":200,
                "headers":{

                },
                "body":{
                    "status":"UP",
                    "appName":"reactuiboilerplate"
                }
            }
        }
    ],
    "metadata":{
        "pactSpecification":{
            "version":"2.0.0"
        }
    }
}
```

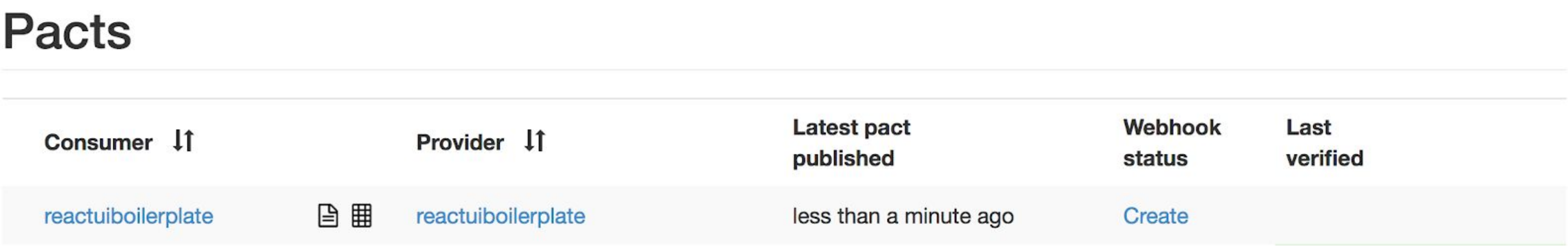
- Publish the pact files to the pact broker.

```
var pact = require('@pact-foundation/pact-node');
let pactBrokerURL = process.env.ENV_PACT_BROKER_URL;
var opts = {
    pactFilesOrDirs: ['test/unit/reports/pacts/'],
    pactBroker: pactBrokerURL,
    consumerVersion: '2.0.0'
};
```

```
pact.publishPacts(opts).then(function () {
    console.log('********** Pact Published **********');
});
```

NPM script :

```
"pact-publish": "ENV_PACT_BROKER_URL=http://172.17.65.26 node test/unit/utils/pact-publish.js"
```



Pacts

Consumer ⇅		Provider ⇅	Latest pact published	Webhook status	Last verified
reactuiboilerplate	🗎 ▦	reactuiboilerplate	less than a minute ago	Create	

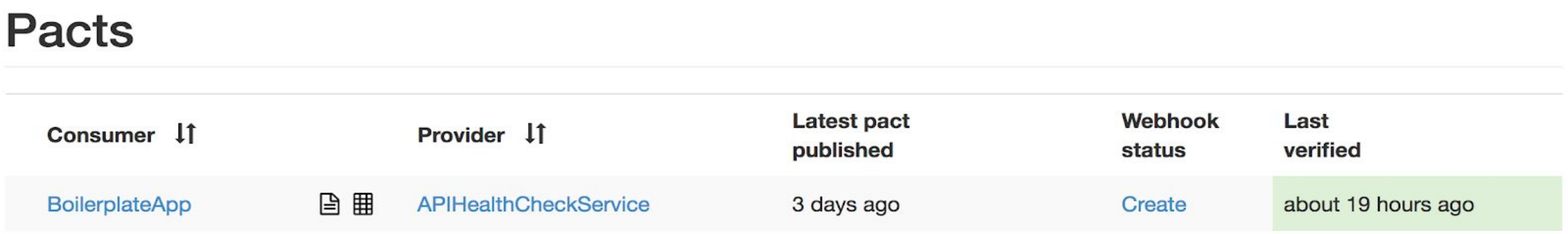
Provider

- Provider will get the pact files from pact broker and verifies pacts(provided the application is up and running).
- Pact verification status will be published to pact broker.

```
const { Verifier } = require('@pact-foundation/pact');
let pactBrokerURL = process.env.ENV_PACT_BROKER_URL;
describe('Pact Verification', () => {
    it('should validate the expectations of Matching Service', function () {
        let opts = {
            provider: 'reactuiboilerplate',
            providerBaseUrl: 'https://localhost:8080',
            pactBrokerUrl: pactBrokerURL,
            publishVerificationResult: true,
            providerVersion: '2.0.0'
        };
        return new Verifier().verifyProvider(opts)
        .then(output => {
            console.log('********** Pact Verification Complete! **********', output);
        });
    });
});
```

NPM script :

```
"pact-verify-provider": "ENV_PACT_BROKER_URL=http://172.17.65.26 ./node_modules/.bin/mocha --reporter
mochawesome --reporter-options
reportDir=test/cdc/reports,enableCode=false,reportFilename=index,reportPageTitle='CDC Pact
Verify',inlineAssets=true --require ./test/unit/utils/helper.js --timeout 10000 test/cdc/provider.spec.js"
```



Pacts

Consumer	Provider	Latest pact published	Webhook status	Last verified
BoilerplateApp	APIHealthCheckService	3 days ago	Create	about 19 hours ago

CDC testing in CI pipeline

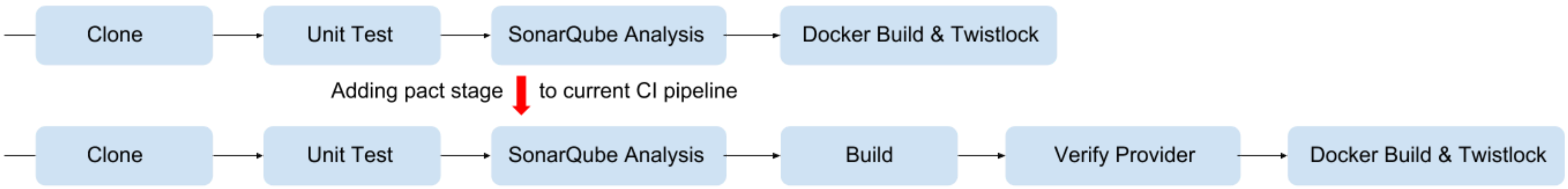
Consumer pipeline :

- Pact publish script needs to be added to the unit test stage

Provider pipeline :

- Current build stage has build the application, create the docker image and push it to registry. It will be split into two, build(creates jar or bundle files) and Docker build
- Verify provider stage will be added after the build stage.
- Verify provider stage will
    - Runs the jar or bundle file created in build stage

    - Executes the verify provider script and publishes the status to pact broker

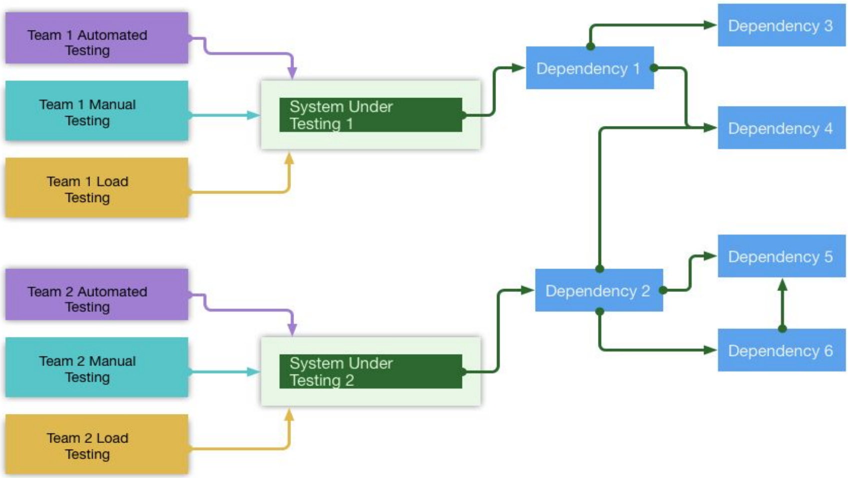


SERVICE VIRTUALIZATION WITH MOUNTEBANK

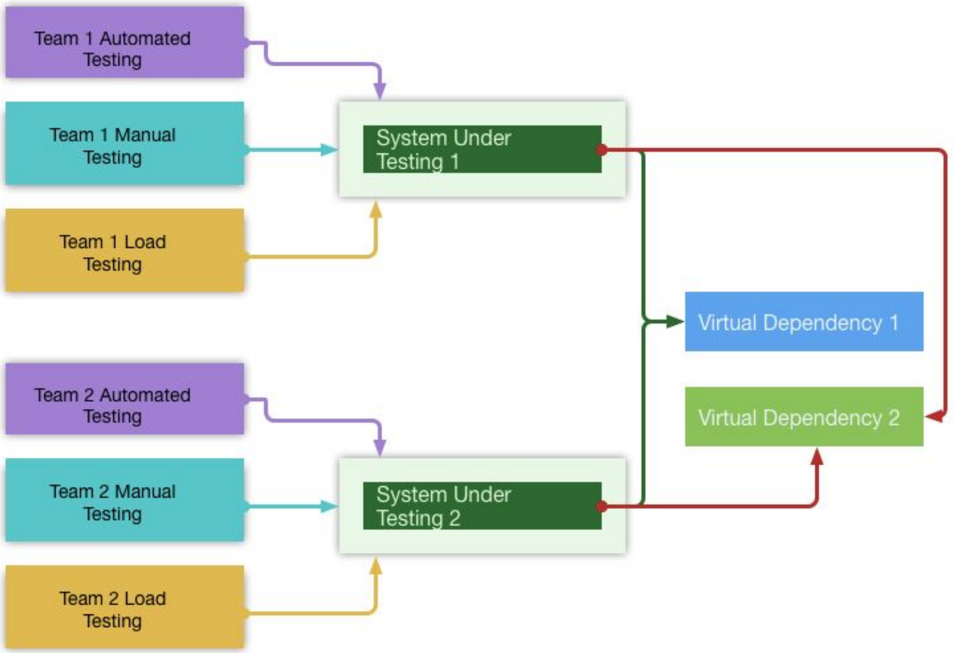
Using Service Virtualization and Microservices

- Running a test in a shared environment means that tests may pass or fail for reasons that have nothing to do with either the service that we are testing or the tests themselves. They could fail due to resource contention with other teams who are touching the same data, or overwhelming the server resource’s of a shared service. They could fail due to environmental instability. They could fail, or be nearly impossible to write to begin with, due to an inability to get consistent test data set up in all the services.
- **End-to-end testing introduces several problems of coordination :** It turns out that this problem has already been solved by reducing the dependency using service virtualization.Service virtualization and microservices are technology techniques being

used to drive a modern software development model to lower cost and increase the speed of software deployment. Rather than depending on a single team to create monolithic application, this model integrates teams to work with multiple interdependent software components.There is ample evidence that the trend toward distributed, concurrent development and testing methodologies, which are supported by service virtualization and microservices, will become increasingly important in the software development lifecycle.



- **Testing Using Virtual Service :** As service virtualization emulates the behavior of software components it helps us to remove the following dependency system constraints.
    - Dependent system services are not ready to consume
    - Controlled by a third-party or partner
    - Difficult to provision or configure in a test environment
    - Needed for simultaneous access by different teams with varied test data setup and other requirements
    - Environment shared with other teams.
    - As a whole, Service Virtualization helps us as follows.
        - Virtualized services to cover functional and performance scenarios.
        - Isolated and repeatable functional and performance tests
        - Faster and independent delivery cycles



Introducing Mountebank

- Mountebank is the first tool to provide cross-platform,multiprotocol test doubles over the wire.Mountebank, a lightweight tool for stubbing and mocking HTTP, HTTPS, SMTP, and TCP.Test Double is a generic term for any case where you replace a production object for testing purposes.
- Setting up is as simple as pointing our microservices to mountebank test double endpoint, instead of the real dependency and pretend as these services are real services with traditional stubs and mocks. This helped us to create virtual environments where we ran product load capacity without any real dependency services.

How it Works:

Mountebank uses imposters to act as on-demand test doubles. Our test communicates to mountebank over http using the api to set up stubs, record and replay proxies, and verify mock expectations. In the typical use case, each test will start an imposter during test setup and stop an imposter during test teardown.

Advantages

- Helps us to create mock responses for HTTP, HTTPS, SMTP, and TCP.
- Mountebank can be configured to give random responses using predicate and filters.
- Custom client scripts can be executed by dynamically injecting custom Java scripts.
- The results can be asserted and matched with certain predefined response to validate minimal unit cases.

Installation

- Installation requires requires node.js v4 or higher
- Run `npm install -g mountebank` to install mountebank

Starting Mountebank

- Mountebank can be started using `mb` command.
- By default, mountebank listens on port 2525, but that's not the port that your imposters (test doubles) will listen on.

Mountebank Terminologies

- `Imposters` - The imposter is the most integral part of mountebank. An imposter is associated with a port number and a protocol. It contains within itself the response we want to return,and other related information.
- `Response -` Defines status code, headers and body.
- `Predicate -` Conditions to check request to match some criteria based on which response will be returned
- `Stub -` Also called Imposter is a Collection of predicates and responses for simulating an API.
- `Behavior - wait -` Adds latency to a response by waiting a specified number of milliseconds before sending the response.

```
{
    "port": 9999,
    "protocol": "https",
    "stubs": [
        {
            "responses": [
                {
                    "is": {
                        "statusCode": 200,
                        "body": "[{\"productInfo\": {\"hairToolType\":\"\", \"hairType\":\"Dry\",
\"fsa_cd\":\"0\", \"storeInv\":\"instock\"}}]"
                    },
                    "_behaviors": {
                        "wait": 500
                    }
                }
            ],
            "predicates": [
                {
                    "and": [
                        {
                            "equals": {
                                "method": "POST"
                            }
                        },
                        {
                            "contains": {
                                "path": "/productsearch/v1/products/search"
                            }
                        },
                        {
                            "contains": {
                                "body": "shampoo"
                            }
                        }
                    ]
                }
            ]
        }
    ]
}
```

Using Predicates to Send Different Responses

- Mountebank helps us to create virtual imposters that pretends to be a real imposter using predicates. Predicates is a condition that determines whether a given stub is responsible for responding. Each stub can have 0 or more predicates.
- Each predicate object contains one or more of the request fields as keys. Predicates are added to a stub in an array, and all predicates are AND'd together. The following predicate operators are allowed:

Operator	Description
equals	The request field matches the predicate
deepEquals	Performs nested set equality on the request field, useful when the request field is an object (e.g. the query field in http)
contains	The request field contains the predicate
startsWith	The request field starts with the predicate
endsWith	The request field ends with the predicate
matches	The request field matches the JavaScript regular expression defined with the predicate.
exists	If true, the request field must exist. If false, the request field must not exist.
not	Inverts a predicate
or	Logically or's two predicates together
and	Logically and's two predicates together
inject	Injects JavaScript to decide whether the request matches or not.

LOAD TESTING WITH ARTILLERY

Installation

- Artillery is written in Node JS. Node 7 is recommended for running Artillery, but any version above 4 will work.
- To install, run `npm install -g artillery`

Configuration

- An Artillery script is composed of two sections: `config` and `scenarios`.
- `config :`
    - `target :` It's the base URL for all requests for an HTTP application
    - `environments :` List of environments, and associated target URLs
    - `phases :` Duration of the test and frequency of requests
    - `payload :` Used for importing variables from a CSV file
    - `defaults :` Set default headers that will apply to all HTTP requests
    - `tls :` Configure how Artillery handles self-signed certificates

- `scenarios :` A scenario is a sequence of steps that will be run sequentially which represents a typical sequence of requests sent by a user of an application. A scenario definition is an object which must contain a `flow` attribute and may contain a number of other attributes.
    - `flow :` It is an array of operations that a virtual user performs, e.g. GET and POST requests for an HTTP-based application
    - `name :` Allows to assign a descriptive name to a scenario, e.g. "search for a product and get its details"
    - `weight :` Allows for the probability of a scenario being picked by a new virtual user to be "weighed" relative to other scenarios.
        - If you have three scenarios with weights 1, 2, and 5, the scenario with the weight of 2 is twice as likely to be picked as the one with the weight of 1, and 2.5 times less likely than the one with weight of 5.
            - scenario 1: 1/8 = 12.5% probability of being picked
            - scenario 2: 2/8 = 25% probability
            - scenario 3: 5/8 = 62.5% probability

Configuration in Detail

config.phases :

- A load phase defines how many new virtual users will be generated in a time period. For example, a typical performance test will have a gentle warm up phase, followed by a ramp up phase which is then followed by a maximum load for a duration of time.
- `config.phases` is an array of phase definitions that Artillery goes through sequentially. Four kinds of phases are supported:
    - A phase with a duration and a constant **arrival rate** of a number of new virtual users per second.
    - A linear **ramp up** phase where the number of new arrivals increases linearly over the duration of the phase.
    - A phase which generates a fixed **count** of new arrivals over a period of time.
    - A **pause** phase which generates no new virtual users for a duration of time.

CONSTANT ARRIVAL RATE : Create 50 virtual users every second for 5 minutes.

```
config:
  target: "https://m-perf.walgreens.com"
  phases:
    - duration: 300
      arrivalRate: 50
```

RAMP UP FOLLOWED BY A CONSTANT ARRIVAL RATE : Ramp up arrival rate from 10 to 50 over 2 minutes, followed by 10 minutes at 50 arrivals per second.

```
config:
  target: "https://m-perf.walgreens.com"
  phases:
    - duration: 120
      arrivalRate: 10
      rampTo: 50
      name: "Warm up the application"
    - duration: 600
      arrivalRate: 50
      name: "Sustained max load"
```

FIXED COUNT OF ARRIVALS : Create 20 virtual users in 60 seconds (approximately one every 3 seconds)

```
config:
  target: "https://m-perf.walgreens.com"
  phases:
    - duration: 60
      arrivalCount: 20
```

DO-NOTHING PHASE :

```
config:
  target: "https://staging.example.com"
  phases:
    - pause: 60
```

config.environments :

- Re-use a load testing script across multiple environments (e.g. dev, staging, and perf). `artillery run -e staging my-script.yml`

```
config:
  target: "https://m-perf.walgreens.com"
  phases:
```

```
  - duration: 10
   arrivalRate: 1
 environments:
  dev:
   target: "https://m-int1.walgreens.com"
   phases:
    - duration: 120
     arrivalRate: 10
  staging:
   target: "https://m-qa2.walgreens.com"
   phases:
    - duration: 1200
     arrivalRate: 20
scenarios:
 - ...
```

config.payload :

- In some cases it is useful to be able to inject data from external files into the test scenarios. For example, we might have a list of product ids that we want to use to test product details page with different products.
- Payload files are in the CSV format and Artillery allows us to map each of the rows to a variable name that can be used in scenario definitions

```
config:
 target: 'https://m-perf.walgreens.com'
 payload:
   path: "./productslist.csv"
   fields:
    - "productid"
 phases:
  - duration: 600
   arrivalRate: 20
   name: "Steady Test"
scenarios:
```

```
- name: "Product Details UI"
  flow:
    - get:
        url: "/store/c/aussie-mega-moist-shampoo/ID=prod{{productid}}-product"
```

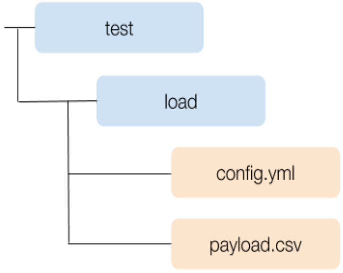
NPM Scripts

```
"load-test": "npm install -g artillery && artillery run -k -o test/load/reports/report.json
test/load/config.yaml"
"load-test-report": "artillery report -o test/load/reports/index.html test/load/reports/report.json"
```

Continuous Integration

config.yml

```
config:
  target: 'https://m-perf.walgreens.com'
  payload:
      path: "./productslist.csv"
      fields:
        - "productid"
  phases:
    - duration: 600
      arrivalRate: 20
      name: "Steady Test"
    - duration: 600
      arrivalRate: 200
      name: "Burst Phase"
scenarios:
  - name: "Product Details UI"
    flow:
      - get:
          weight: 7
          url:
"/store/c/free-%26-clear-shampoo/ID=prod{{productid}}-product?reactjs=true"
  - name: "Product Details API"
```



```
flow:
  - get:
      weight: 3
      url: "/vpd/v1/products/{{productid}}/(PriceInfo+Inventory+ProductDetails)"
```

payload.csv

```
productid
1664357
6274954
6330400
6017176
6038953
6038952
```

- Config and payload files will be checked into the microservice repository.
- Using build slave image(since it just needs node js version > 4) to run the Artillery load test.
- Load test job
  - brings up this Jenkins slave image
  - clones the microservice repository
  - executes Artillery test with the given config/payload file
  - publishes the HTML report

UI PERFORMANCE TESTING WITH SITESPEED.IO

Introduction

- Sitespeed.io is the complete toolbox to test the web performance of your website.
- Test websites using real browsers, simulating real users connectivity and collect important user centric metrics like Speed Index and First Visual Render.
- Analyse how the page is built and give feedback how we can make it faster for the end user.
- Generates HAR file which can be used for debugging.
- Visual rendering of the page can be recorded as a video.
- Browser support : Firefox, Chrome and Chrome on Android.

Installation

- You can run sitespeed.io using our Docker containers or using NodeJS.
- Docker
  - Prerequisites: Docker.
  - Prerequisites: NodeJS LTS, npm, chrome and firefox.
  - No additional setup needed to run sitespeed.io
  - Docker image has sitespeed.io, Chrome, Firefox and Xvfb.

```
docker run --rm -v "$(pwd)":/sitespeed.io sitespeedio/sitespeed.io https://www.walgreens.com -b firefox
```

- Node JS
  - Prerequisites: Latest NodeJS LTS, npm, chrome and firefox.
  - `npm install sitespeed.io -g`

Configuration

- Below configuration is to execute sitespeed.io test in headless chrome. Please refer https://www.sitespeed.io/documentation/sitespeed.io/configuration/ for more details.

```
{
    "browsertime": {
        "iterations": 2,
        "speedIndex" : true,
        "connectivity": {
            "profile": "3gfast"
        },
        "xvfb": true,
        "chrome": {
            "args": [
                "no-sandbox"
            ]
        }
    },
    "html": {
        "showAllWaterfallSummary": true
    },
    "outputFolder": "test/performance/reports",
    "mobile": true,
    "budget": {
        "configPath": "test/performance/budget.json",
        "output": "tap"
    }
}
```

Performance Budget

- Defining SLA on different metrics like no of js/css/image requests should be made from the page.
- If we run sitespeed.io configured with a budget, the script will exit with an exit status > 0 if the budget fails. It will log all budget items regardless if they pass or fail and generate a HTML report.

```
{
    "browsertime.pageSummary": [
        {
            "metric":"statistics.visualMetrics.SpeedIndex",
            "max":1000
        }
    ],
    "coach.pageSummary": [],
    "pagexray.pageSummary": [
        {
            "metric": "statistics.timings.SpeedIndex",
            "max": 2000
        },
        {
            "metric": "contentTypes.css.requests",
            "max": 10
        },
        {
            "metric": "contentTypes.javascript.requests",
            "max": 10
        },
        {
            "metric": "contentTypes.image.requests",
            "max": 120
        }
    ]
}
```

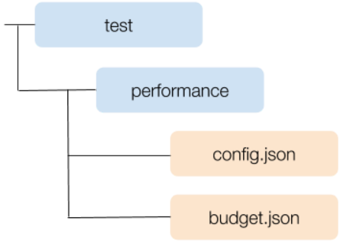
- Output of budget result can be created as JUnit XML or TAP file

NPM Scripts

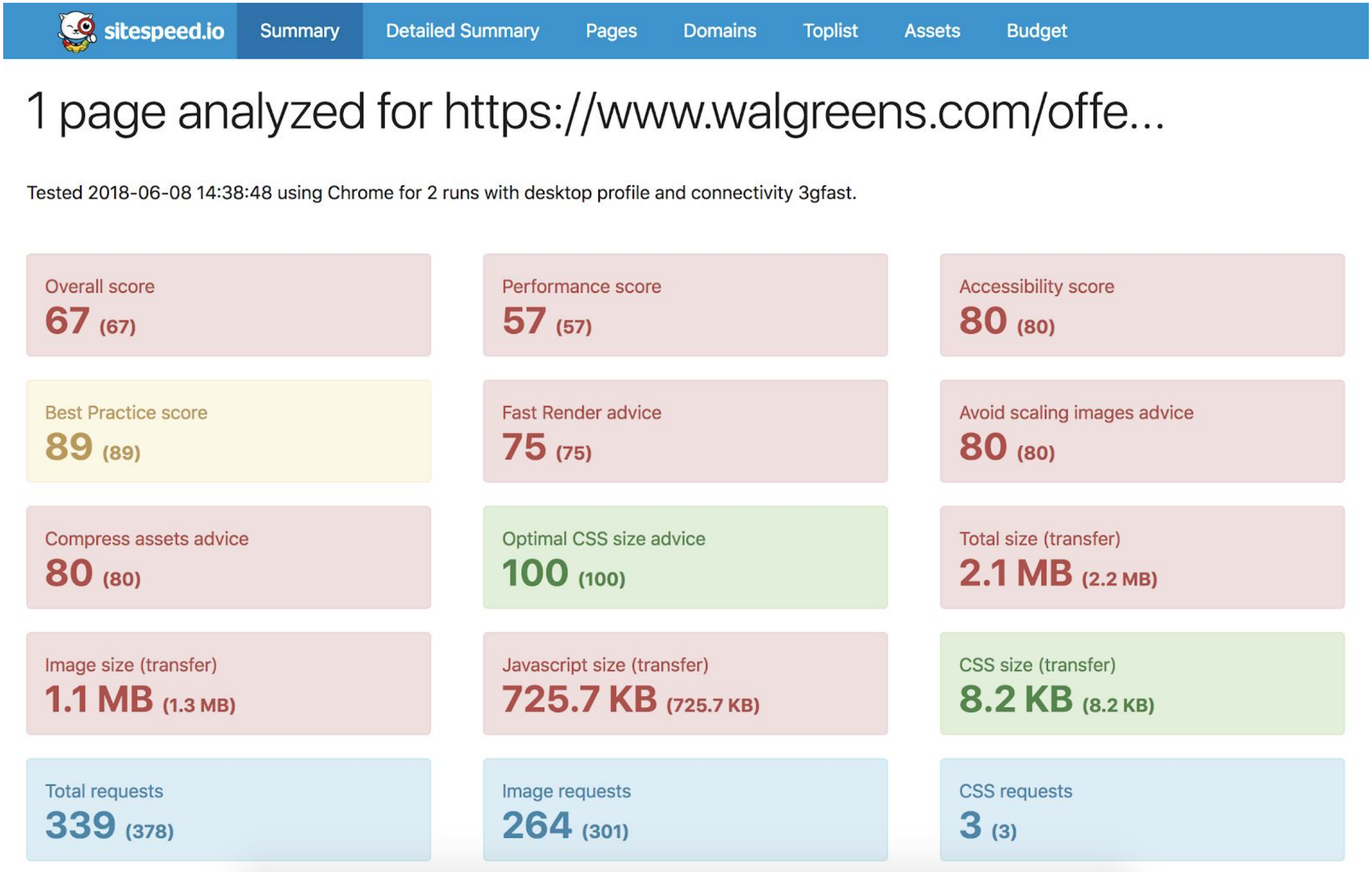
```
"ui-perf-test-dev": "/usr/src/app/bin/sitespeed.js --config=test/performance/config.json
https://172.17.65.222/reactuiboilerplate/v1/page"
```

Continuous Integration

- Config and budget files will be checked into the microservice repository.
- Created custom Jenkins slave image with sitespeed.io image in it.
- UI performance test job
  - brings up this Jenkins slave image
  - clones the microservice repository
  - executes sitespeed.io test with the given config/budget file
  - publishes the HTML and TAP report



Walgreens



1 page analyzed for https://www.walgreens.com/offe...

Tested 2018-06-08 14:38:48 using Chrome for 2 runs with desktop profile and connectivity 3gfast.

Overall score
67 (67)

Performance score
57 (57)

Accessibility score
80 (80)

Best Practice score
89 (89)

Fast Render advice
75 (75)

Avoid scaling images advice
80 (80)

Compress assets advice
80 (80)

Optimal CSS size advice
100 (100)

Total size (transfer)
2.1 MB (2.2 MB)

Image size (transfer)
1.1 MB (1.3 MB)

Javascript size (transfer)
725.7 KB (725.7 KB)

CSS size (transfer)
8.2 KB (8.2 KB)

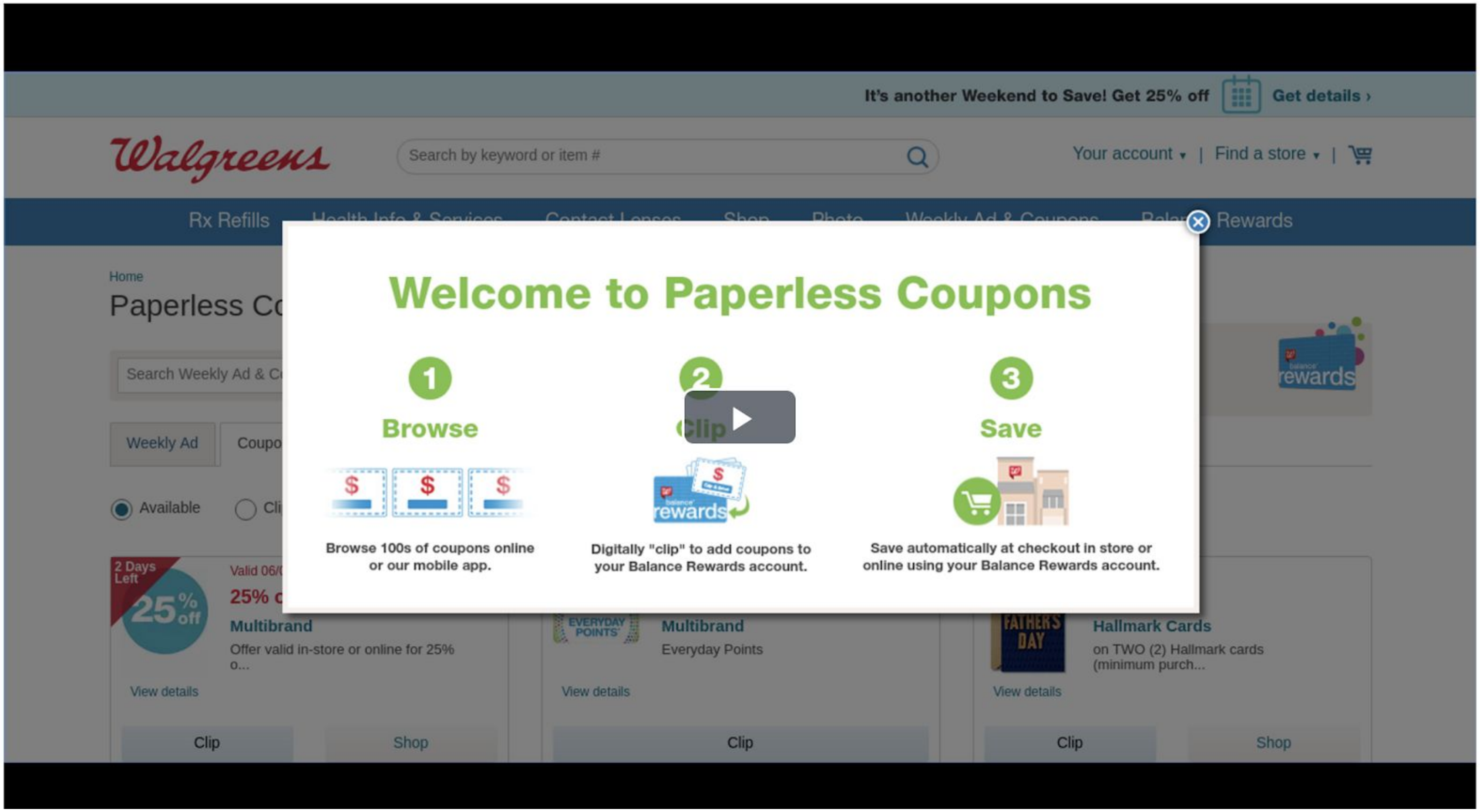
Total requests
339 (378)

Image requests
264 (301)

CSS requests
3 (3)

sitespeed.io Summary Detailed Summary Pages Domains Toplist Assets Budget

Video



It's another Weekend to Save! Get 25% off Get details ›

Walgreens
Search by keyword or item #
Your account ▾ | Find a store ▾ |
Rx Refills Health Info & Services Contact Lenses Shop Photo Weekly Ad & Coupons Balance Rewards
Home
Paperless Co
Search Weekly Ad & Co
Weekly Ad
Coupo
Available
Cli

Welcome to Paperless Coupons

1 Browse

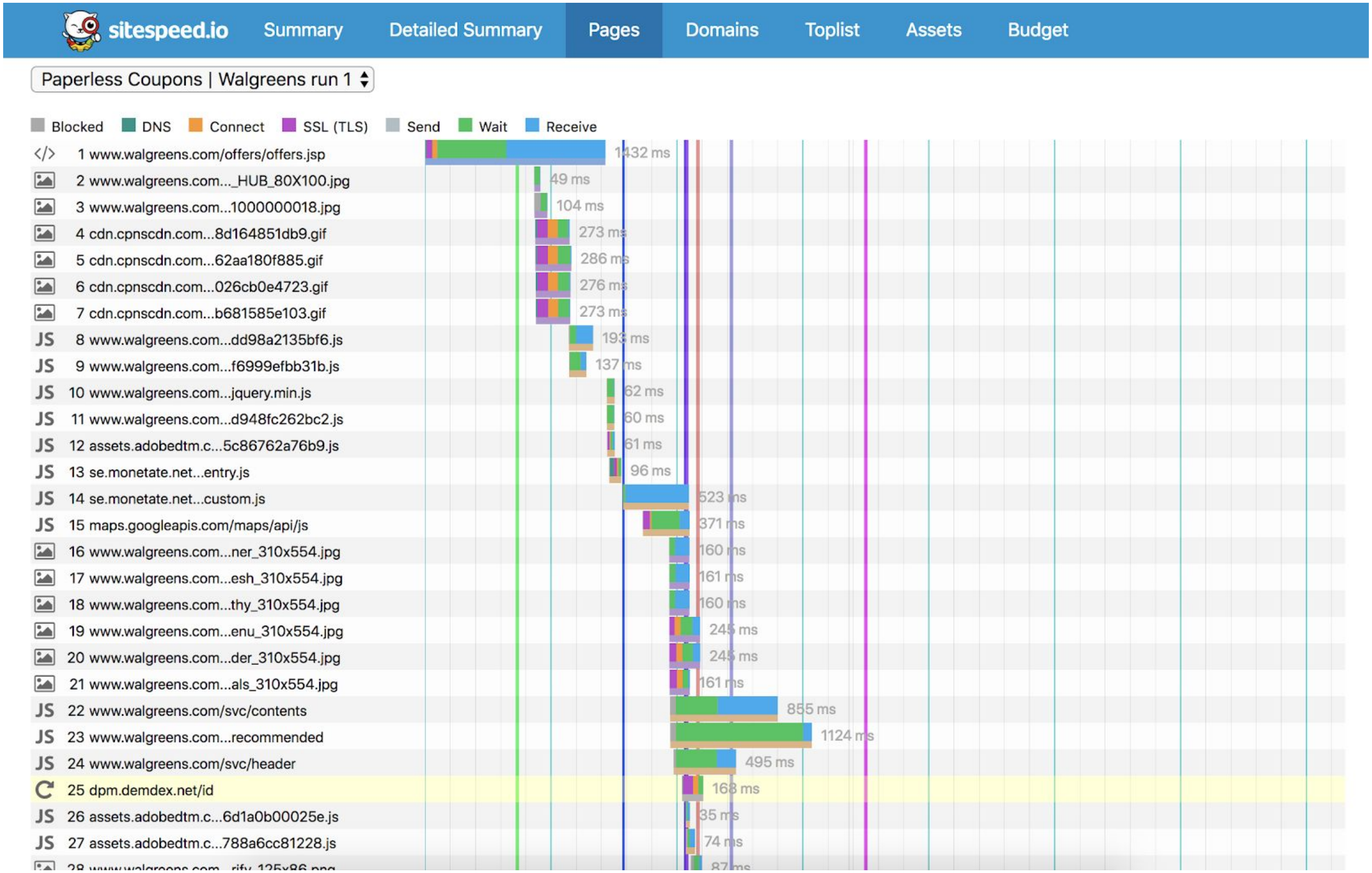
Browse 100s of coupons online or our mobile app.

2 Clip

Digitally "clip" to add coupons to your Balance Rewards account.

3 Save

Save automatically at checkout in store or online using your Balance Rewards account.

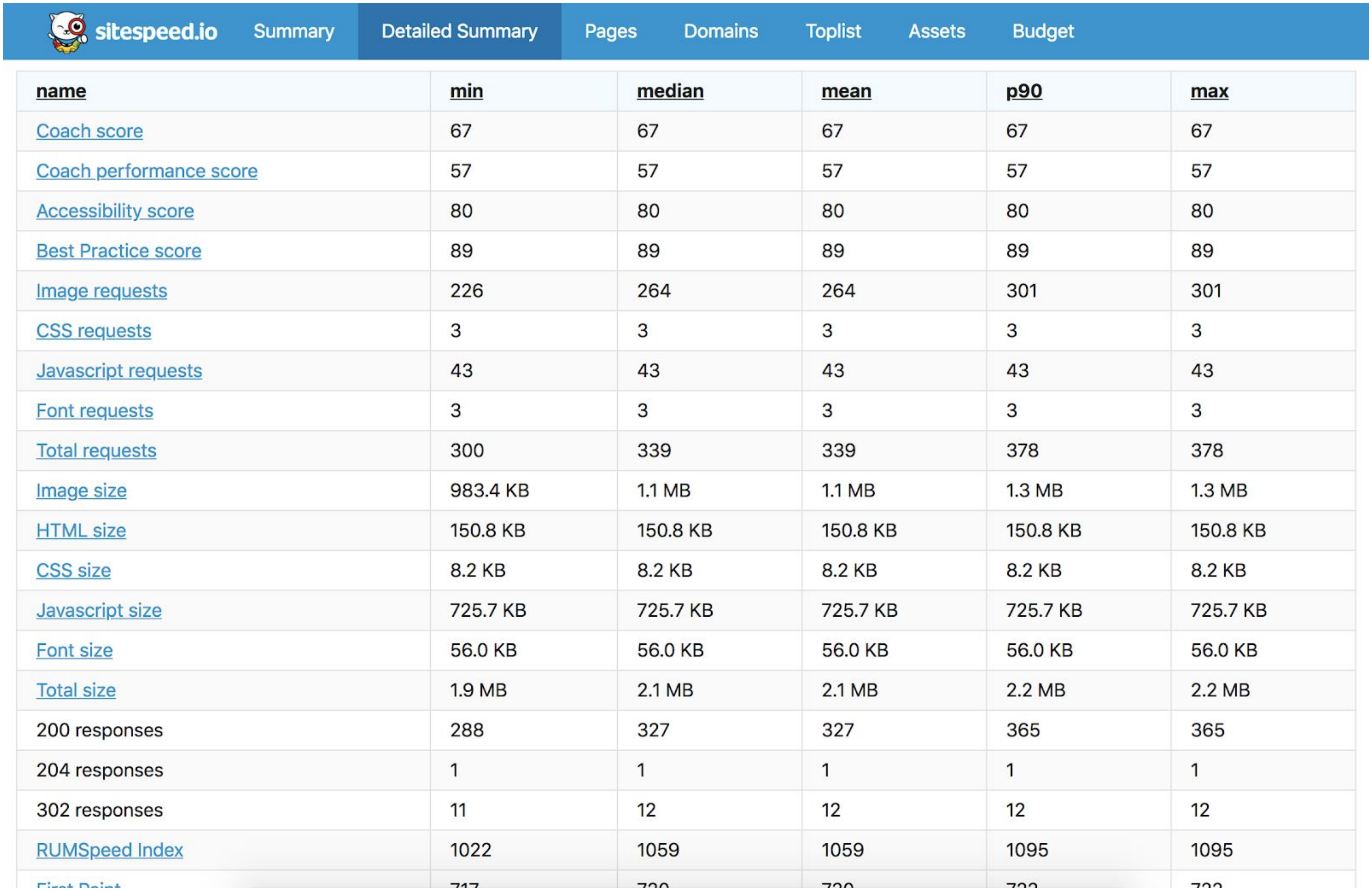


Paperless Coupons | Walgreens run 1

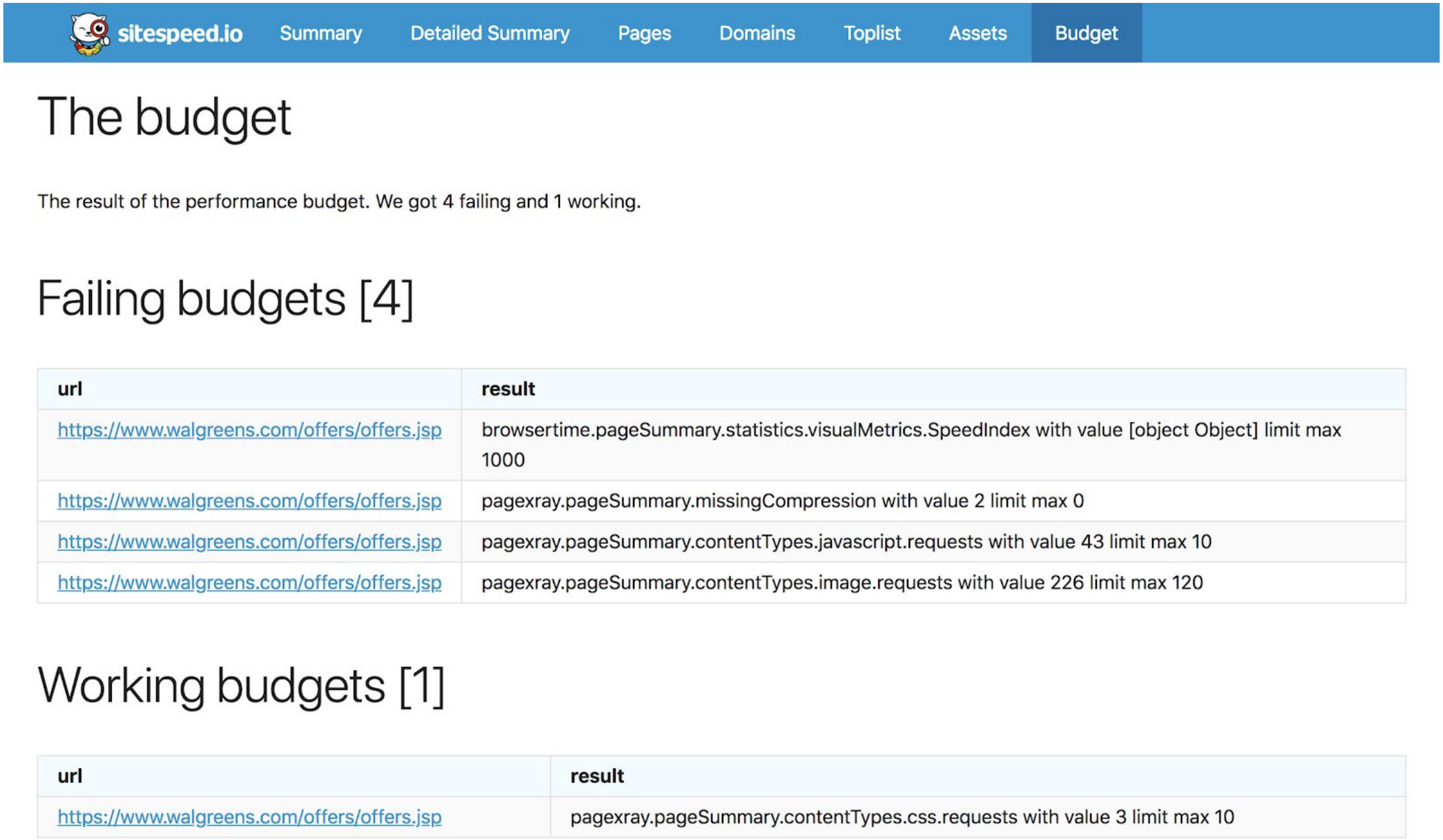
Blocked DNS Connect SSL (TLS) Send Wait Receive

1 www.walgreens.com/offers/offers.jsp 1432 ms
2 www.walgreens.com..._HUB_80X100.jpg 49 ms
3 www.walgreens.com...1000000018.jpg 104 ms
4 cdn.cpnscdn.com...8d164851db9.gif 273 ms
5 cdn.cpnscdn.com...62aa180f885.gif 286 ms
6 cdn.cpnscdn.com...026cb0e4723.gif 276 ms
7 cdn.cpnscdn.com...b681585e103.gif 273 ms
8 www.walgreens.com...dd98a2135bf6.js 193 ms
9 www.walgreens.com...f6999efbb31b.js 137 ms
10 www.walgreens.com...jquery.min.js 62 ms
11 www.walgreens.com...d948fc262bc2.js 60 ms
12 assets.adobedtm.c...5c86762a76b9.js 61 ms
13 se.monetate.net...entry.js 96 ms
14 se.monetate.net...custom.js 523 ms
15 maps.googleapis.com/maps/api/js 371 ms
16 www.walgreens.com...ner_310x554.jpg 160 ms
17 www.walgreens.com...esh_310x554.jpg 161 ms
18 www.walgreens.com...thy_310x554.jpg 160 ms
19 www.walgreens.com...enu_310x554.jpg 245 ms
20 www.walgreens.com...der_310x554.jpg 245 ms
21 www.walgreens.com...als_310x554.jpg 161 ms
22 www.walgreens.com/svc/contents 855 ms
23 www.walgreens.com...recommended 1124 ms
24 www.walgreens.com/svc/header 495 ms
25 dpm.demdex.net/id 168 ms
26 assets.adobedtm.c...6d1a0b00025e.js 35 ms
27 assets.adobedtm.c...788a6cc81228.js 74 ms
28 www.walgreens.com...rify_125x86.png 87 ms

Walgreens



name	min	median	mean	p90	max
Coach score	67	67	67	67	67
Coach performance score	57	57	57	57	57
Accessibility score	80	80	80	80	80
Best Practice score	89	89	89	89	89
Image requests	226	264	264	301	301
CSS requests	3	3	3	3	3
Javascript requests	43	43	43	43	43
Font requests	3	3	3	3	3
Total requests	300	339	339	378	378
Image size	983.4 KB	1.1 MB	1.1 MB	1.3 MB	1.3 MB
HTML size	150.8 KB	150.8 KB	150.8 KB	150.8 KB	150.8 KB
CSS size	8.2 KB	8.2 KB	8.2 KB	8.2 KB	8.2 KB
Javascript size	725.7 KB	725.7 KB	725.7 KB	725.7 KB	725.7 KB
Font size	56.0 KB	56.0 KB	56.0 KB	56.0 KB	56.0 KB
Total size	1.9 MB	2.1 MB	2.1 MB	2.2 MB	2.2 MB
200 responses	288	327	327	365	365
204 responses	1	1	1	1	1
302 responses	11	12	12	12	12
RUMSpeed Index	1022	1059	1059	1095	1095
First Paint	717	720	720	722	722



The budget

The result of the performance budget. We got 4 failing and 1 working.

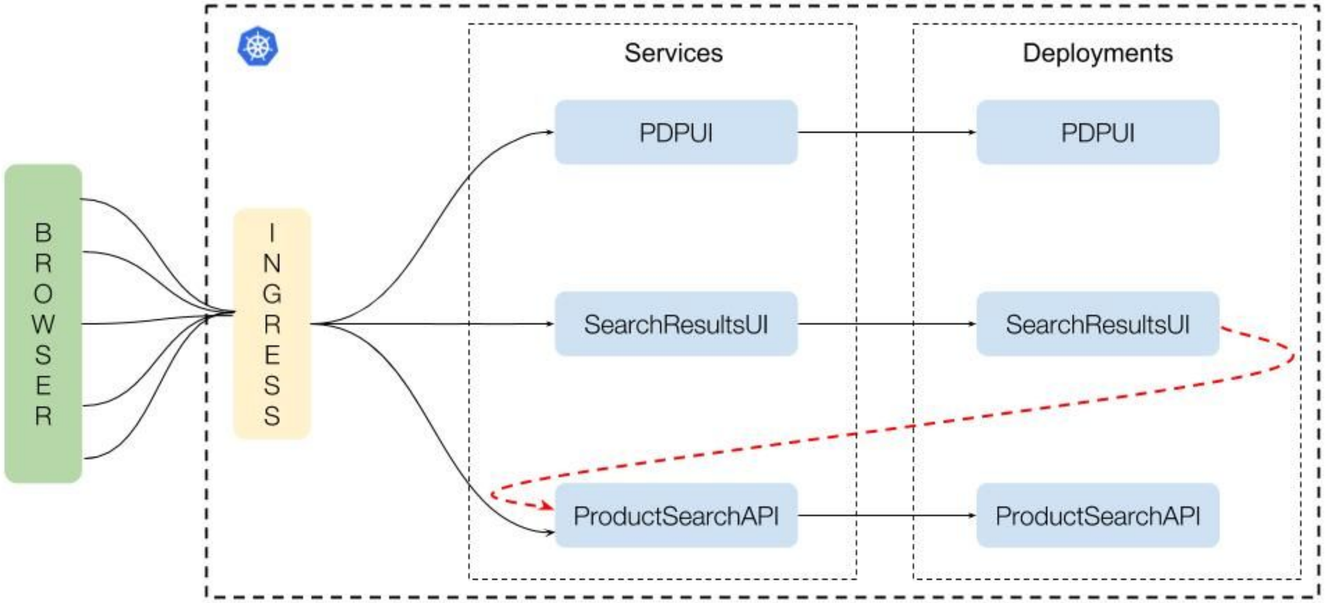
Failing budgets [4]

url	result
https://www.walgreens.com/offers/offers.jsp	browsertime.pageSummary.statistics.visualMetrics.SpeedIndex with value [object Object] limit max 1000
https://www.walgreens.com/offers/offers.jsp	pagexray.pageSummary.missingCompression with value 2 limit max 0
https://www.walgreens.com/offers/offers.jsp	pagexray.pageSummary.contentTypes.javascript.requests with value 43 limit max 10
https://www.walgreens.com/offers/offers.jsp	pagexray.pageSummary.contentTypes.image.requests with value 226 limit max 120

Working budgets [1]

url	result
https://www.walgreens.com/offers/offers.jsp	pagexray.pageSummary.contentTypes.css.requests with value 3 limit max 10

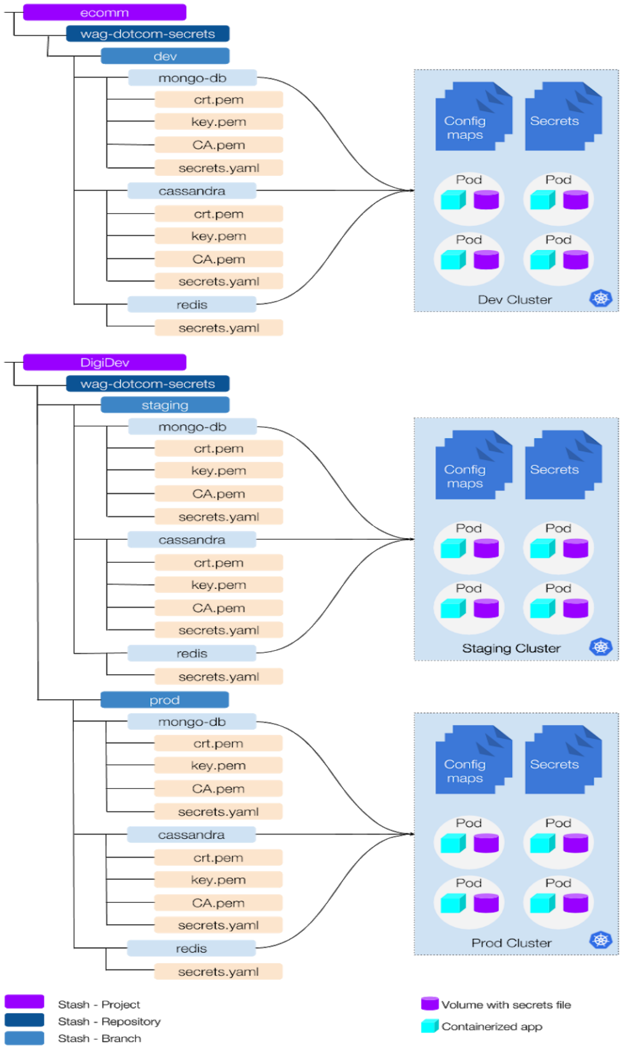
INTERNAL CONNECTIVITY BETWEEN APPLICATIONS



- In the above diagram,
    - Deployments has the containerized apps running
    - A service created for respective deployments
    - Ingress controller has all the routes that points to the respective services
- Application can be accessed from
    - Browser - Using external host(ingress)
    - Server - Applications running in same cluster can connect to each other using internal host.
        - From the above example, we have SearchResultsUI that gets data from `ProductSearchAPI`.
        - By default, Kubernetes will inject host and port of each services as environment variables to the Pods.

- In our case, `ProductSearchAPI` host as **`PRODUCTSEARCHAPI_SERVICE_HOST`** and `ProductSearchAPI` port as **`PRODUCTSEARCHAPI_SERVICE_PORT`**.
- `SearchResultsUI` can use these environment variables to form `ProductSearchAPI` internal URL and get the data.

SECRETS MANAGEMENT IN KUBERNETES



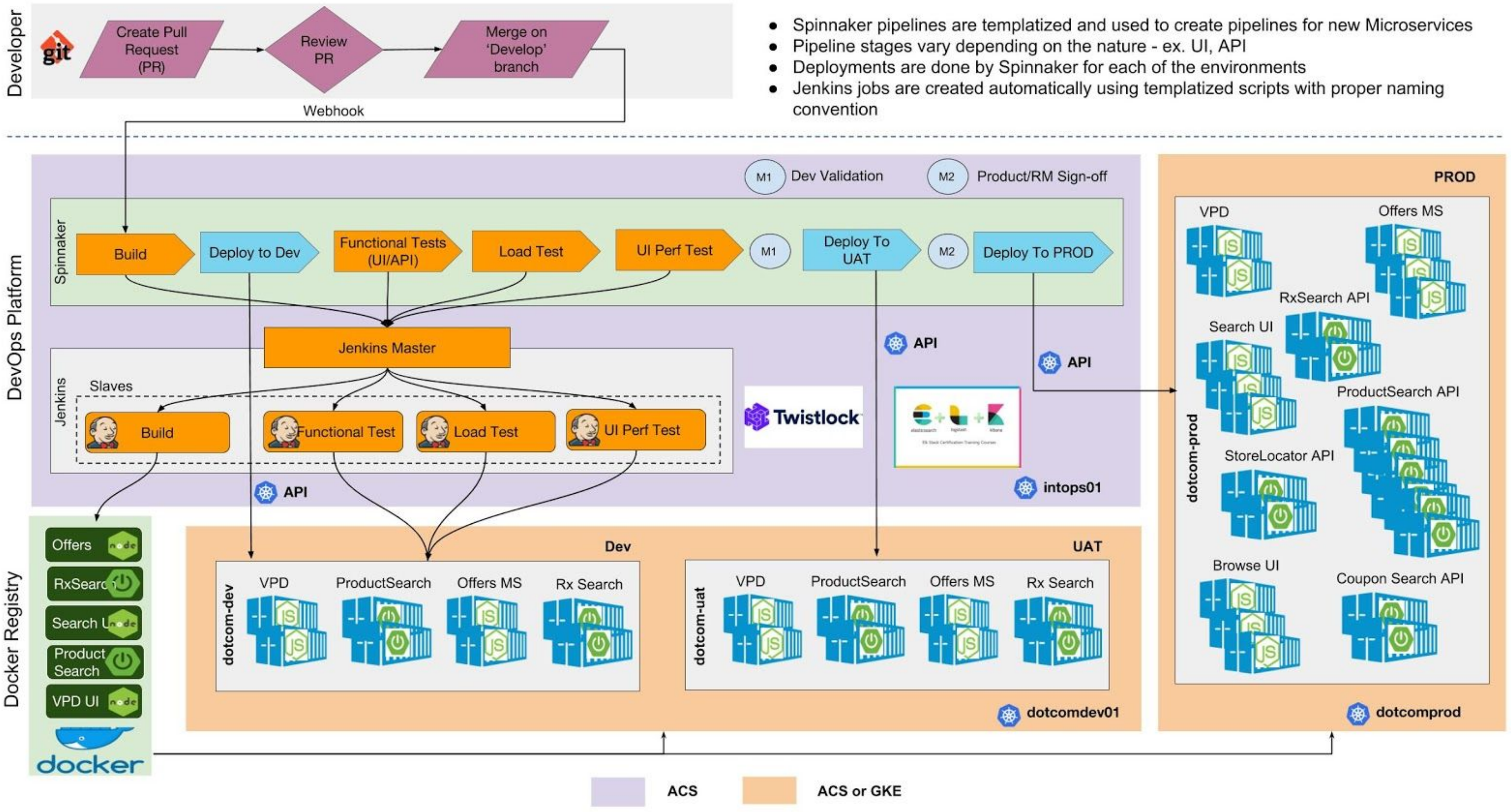
- **wag-dotcom-secrets :** It’s the repository in stash where all the certificates and the secrets yaml files are checked-in.
  - Development environment secrets will be available in `wag-dotcom-secrets` repository under `ecomm` project, so that all the developers will have read access to this repository
  - Staging and production environment secrets will be available in respective branches in `wag-dotcom-secrets` repository under `DigiDev` project. Note : Only selected people will have access to `DigiDev` project.

```
apiVersion: v1
kind: Secret
metadata:
  name: mongodb
data:
  crt.pem: LS0tLS1CRUdJTiBDRVJUSUZJQ0FU...
  key.pem: LS0tLS1CRUdJTiBQUklWQVRFIEtF...
  ca.pem: LS0tLS1CRUdJTiBDRVJUSUZJQ0FUR...
  username: YWRtaW4=
  password: cGFzc3dvcmQ=
```

- **wag-dotcom-secrets/dev/mongo-db/secrets.yaml :**
  - Value of each keys are base64 encoded(Ex : `cat crt.pem | base64` or `echo -n "admin" | base64`)
  - These secrets will be created in respective environments using `kubectl apply -f secrets.yaml` during cluster creation and whenever there is any change in the secrets.

- During the application deployment in ACS cluster, these secrets are mounted as files to the pod. Application should have logic to read these files and consume the data. Kubernetes by default will provide the decoded data.
- For local dev environment, developers need to clone the respective branches(dev) and use the certificates/credentials to connect to appropriate service

CI/CD ARCHITECTURE



- Spinnaker pipelines are templatized and used to create pipelines for new Microservices
- Pipeline stages vary depending on the nature - ex. UI, API
- Deployments are done by Spinnaker for each of the environments
- Jenkins jobs are created automatically using templatized scripts with proper naming convention

JENKINS SLAVE

Jenkins Slave Docker Image Details

- Docker configurations of these slave agents are in DevOps repository
- Docker image of each slave agents are created and pushed to private Docker Registry(ACR)

Jenkins Slave Image	Details
NodeJS Build	**OS** : Ubuntu. **Tools** : Jenkins slave agent base image, node.js, kubectl CLI
Functional Test	**OS** : Ubuntu. **Tools** : Jenkins slave agent base image, node.js, Google chrome latest stable version
Load Test	Uses **Build** slave image
UI Perf Test	**OS** : Ubuntu. **Tools** : Jenkins slave agent base image, sitespee.io

Master - Slave Configuration

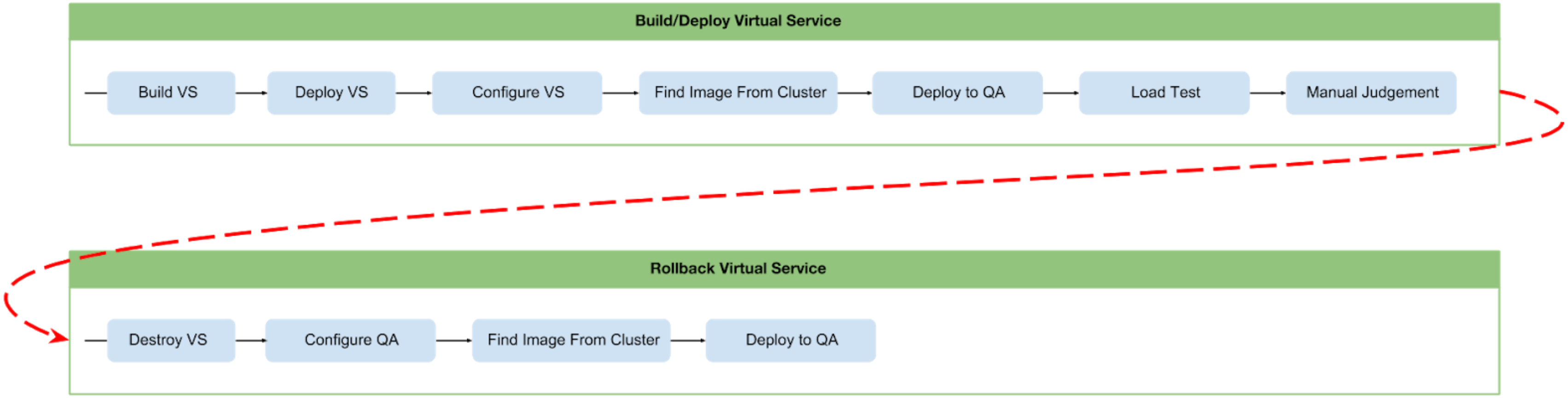
- Using `kubernetes` Jenkins plugin to configure the slave agent pod(container name, docker image to use and etc.,)
- Stages in microservice Jenkinsfile will be wrapped by `node` and `container` in which these stages needs to be executed.

```
node ('wag-dotcom-nodejs-build') {
        container('wag-dotcom-nodejs-build') {
                stage('Clone') {
                        ...
                }
                stage(Validation) {
                        ...
                }
                stage('Build') {
```

```
                                ...
                        }
                }
        }
```

- When Jenkins pipeline starts and if the `node` name in the pipeline matches with `Kubernetes` plugin configuration, the plugin will pull appropriate slave image from ACR and runs the image as a pod in ACS cluster where jenkins is running. Once the pod is up, the stages defined in the pipeline will be executed in the pod.
- Pod(Jenkins slave) will be deleted once all the defined stages are complete.

VIRTUAL SERVICE CI/CD PIPELINE



Virtual service pipeline is not tied with QA pipeline so it has to be triggered manually. Pipeline has been split into two

- Build/deploy virtual service pipeline and execute the load test.
- Rollback virtual service and point to real APIs once load test is done. And this is configured as downstream pipeline of CI/CD pipeline.

Build/Deploy Virtual Service Pipeline Stages

Build VS	● Clones the develop branch of the microservice ● Builds the virtual service and creates docker image. ● Docker image will then be pushed to Azure Container Registry. Different versions of image will not be maintained, it will always be pushed with latest tag. ● Microservice Prerequisites : ○ ReactJS/Node JS : `pack-vs` npm script in package.json, `deployment/Dockerfile-VS` ○ Springboot : `buildVS` and `createVSDockerfile` gradle task.
Deploy VS	Deploys the virtual service image in Kubernetes. Virtual service deployment is also same as the application so it should be started in port `8080` and as `non-ssl`.
Configure VS	Creates the Kubernetes config map object with `config/env/virtual/config.yaml` that resides in the microservice repository. This config file will have the configuration that points to the deployed virtual service.
Find Image From Cluster	Gets the microservice image that is currently running in the QA cluster.
Deploy to QA	Deploys the image received as part of `Find Image From Cluster` stage pointing to virtual service.
Load Test	Executes the load test script and publishes the report.
Manual Judgement	Decision making stage to destroy virtual service and rollback the microservice deployment to point to real APIs.

Rollback Virtual Service Pipeline Stages

Destroy VS	Destroys the virtual service that is currently running.
Configure QA	Creates the Kubernetes config map object with `config/env/qa/config.yaml` that resides in the microservice repository.
Find Image From Cluster	Gets the microservice image that is currently running in the QA cluster.
Deploy to QA	Deploys the image received as part of `Find Image From Cluster` stage pointing to virtual service.