ENGINEERING AND R&D SERVICES

HCL

Cloud Ready Web Applications with jHipster

WHITE PAPER

2 Engineering experiences

Acceleration | Value | Technology

Table of Contents

Introduction

Web application technology is continuously evolving and we need to adapt to the "new normal" of applications and services being cloud ready, distributed, resilient to failure, API-driven, scalable, and more. This document covers how we can build next generation Java web applications using all the best practices and state-of-the-art frameworks, while still providing enhanced agility to the developer to focus on the core business logic of wiring up the entire architecture using the leading Spring Boot + AngularJS application generator - jHipster. We stumbled upon this in a spring.io blog about a month back, while trying to understand Spring Boot and how to leverage it beyond a basic sample application into a real life web application and tried jHipster out, to experience the power of the tooling. The turnaround time for creating a production ready, secure, customer facing, and distributed web application has just got shorter.

For this evaluation exercise, a sample application has been generated as per the tutorial provided here. With the extremely fast pace of new milestone releases and features being added in this project, only some of the main features are being covered in this document. Do visit the site for the latest updates and full details of the feature set.

Key Architecture Drivers

To understand what jHipster provides, it is best to first understand what we, in the developer community, try to achieve when we build modern Java web applications. Some of the key aspects are highlighted below:

- UI Layer: The UI layer is what makes the first impression to the end user and needs to be easy to transition from a concept to a working model. It should support features like responsive design, multi-device support, and fast load time. This requires the usage of technologies like HTML5/CSS3/Java Script based on the Single Page Application architecture.
- Technology selection for the architecture with high developer productivity.
    - There are some great tools for various layers of a web architecture; but, the key challenge is not only in making the right selection(s), but also in being able to make them work together to provide high developer productivity
    - Deployment of applications in multiple environments from local developer laptops to Cloud environments like Heroku / AWS / Cloudfoundry, etc.

- No vendor lock in: Ability to maintain the code / use Open Source stack. Standards-based development that is flexible to extend or change parts of the framework without a major re-write

- Production readiness
  - Quoting from the "12 Factor App" of building software as a service apps – it can **scale up** without significant changes to tooling, architecture, or development practices
  - Secure , RESTful API centric and stateless architecture
  - Logging, monitoring, and auditing support with an admin console or dashboard
  - Last, but not the least, a strong testing infrastructure for all aspects - UI, services, performance and scalability

The goals of jHipster is to solve these issues as stated on its website.

- A beautiful front-end, with the latest HTML5/CSS3/JavaScript frameworks
- A robust and high-quality back-end, with the latest Java/Caching/Data access technologies
- All automatically wired up, with security and performance in mind
- And great developer tooling, for maximum productivity

What is jHipster?

The technology behind JHipster

The heart of the code generation for jHipster is a scaffolding tool called yeoman .

The goal of the tool is to help *"Leverage the success and lessons learnt from several open-source communities to ensure that the developers use it as intelligently as possible".*

The Yeoman workflow comprises three types of tools for improving productivity and satisfaction when building a web app: the scaffolding tool (yo), the build tool (Grunt, Gulp, etc.), and the package manager (like Bower and npm).

The setup requires you to install these workflow components in your developer environment as a pre-requisite. It also supports a Docker version of the developer environment, with a Dockerfile based on a Ubuntu image.

Creating a jHipster Application

High level steps for creating a jHipster application are illustrated below. The workflow for each step has multiple user input steps, which can be referred to, here.

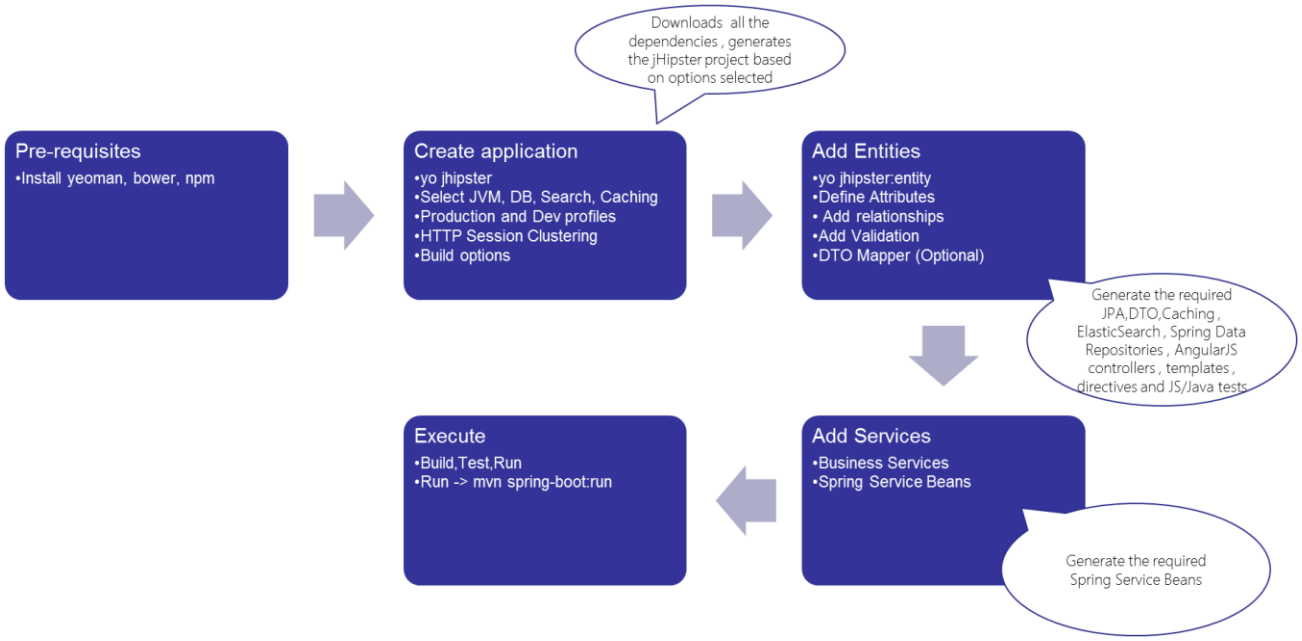


Figure 1: jHipster Project Workflow

As part of the evaluation, we created the jHipster application with the following generation options:

- Java 8 – to get Elasticsearch support (only supported for Java 8)
- Elastic Search
- EhCache enabled – L2 Cache for Hibernate 4.x
- Hazelcast based http session clustering
- MySQL database for both development and production (we could choose H2 for dev. as well)
- No Websockets
- Maven build
- Grunt for frontend build
- No Sass compilation (can be used for advanced CSS users)
- Entities generation options
    - Author 1..* Book
    - Author with link based pagination
    - Book with infinite scroll based pagination support

Client Side Technologies

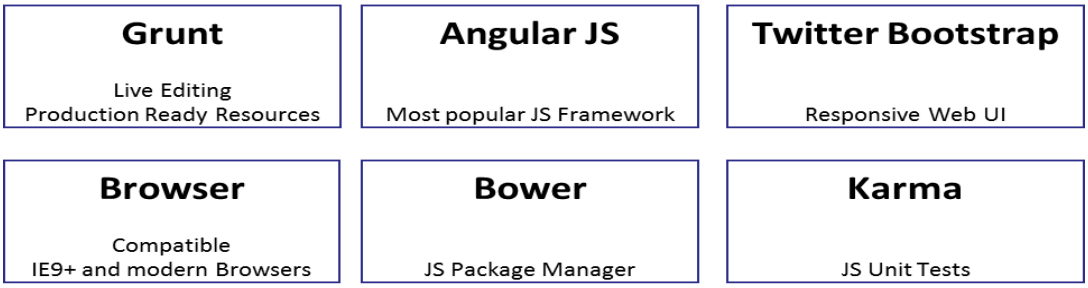


Figure 2: Client Side Technologies

The client side technology stack, as shown above, is the recommended set by many developer communities. However, the key aspect is making these work together in a seamless manner. Some of the key highlights include:

- Increased productivity of client-side Java Script application development and testing: jHipster integrates with BrowserSync , which allows you to have an automatic refresh of all browsers as you modify your HTML/CSS/JavaScript files. This saves you a lot of time as there is no re-build required for testing out the UI changes. The ability to test multiple browsers/devices and have the scroll or click events simultaneously working on all screens is an extremely powerful utility.
- Karma and Phantom JS (headless Web kit) are integrated into the jHipster generated project which helps test your UI without launching the Java backend.
- The UI layer has AngularJS based internalization and is also generated as part of the project. It uses the Angular-Translate module for the same.
- Role Based Access: The UI layer has Angular JS directives for ensuring menu and link availability, and is based on the user context and roles assigned to the user.

Only the ADMIN user role has access to the administration menu.

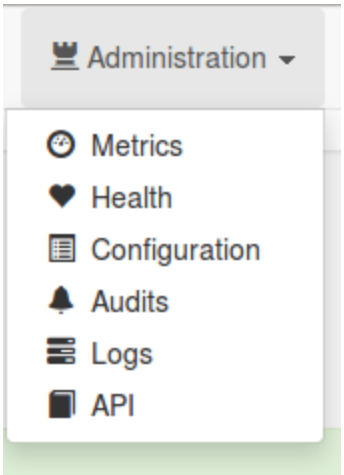


Figure 3: Admin User Access menu

The API level security prevents direct URL access to any of the administrative operations as well.

- Entity related CRUD screens with validation: As entities are added using the jHipster entity generation workflow, Bootstrap CRUD screens/AngularJS controllers with optional pagination and search capability are automatically generated. It wires up with the backend REST API calls from the Spring resource server.

Figure 4: Authors Entity CRUD Screen

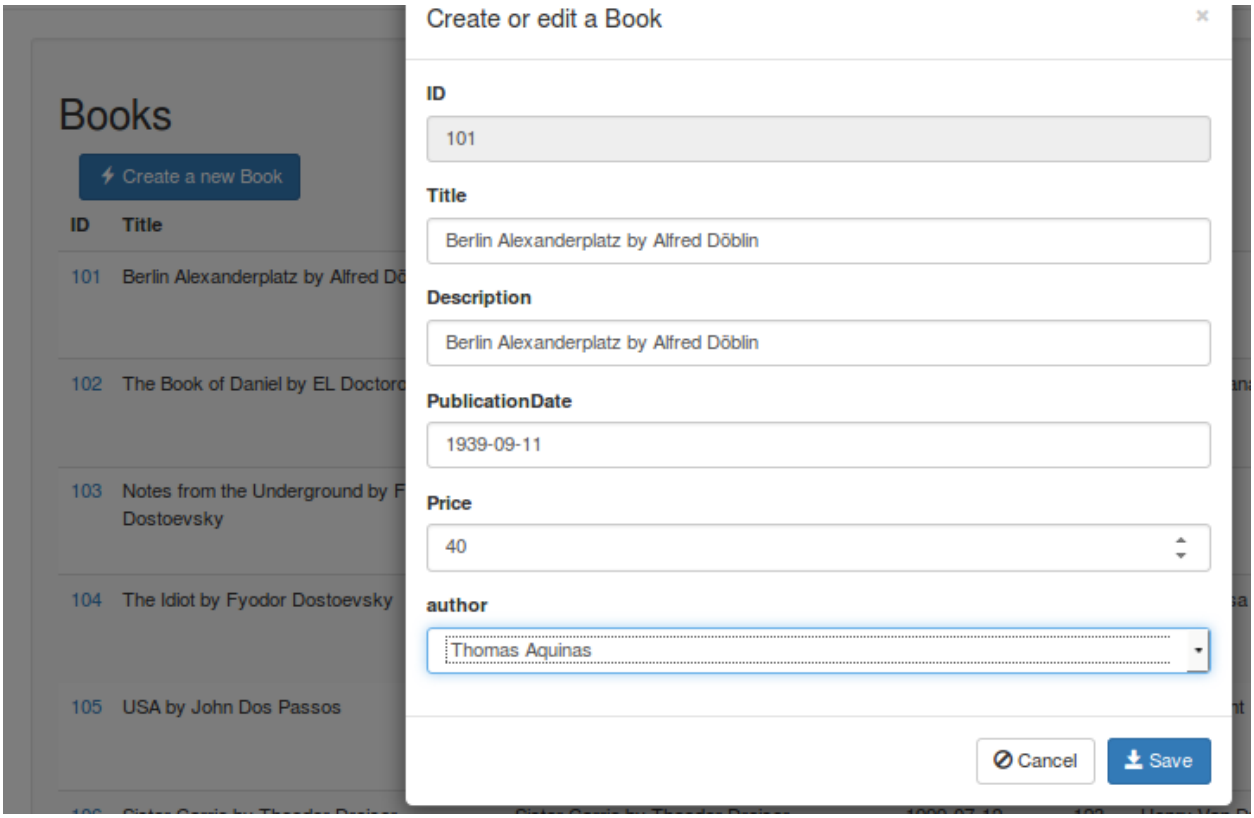


Figure 5: Many to One author drop down selection generation

- The Bower based Java script Package Manager helps the developer quit worrying about version numbers getting entangled in the source code. All client side dependencies are taken care of in the bower.json file, making it much faster to upgrade a version of the library without affecting the source code.

Startup Screen

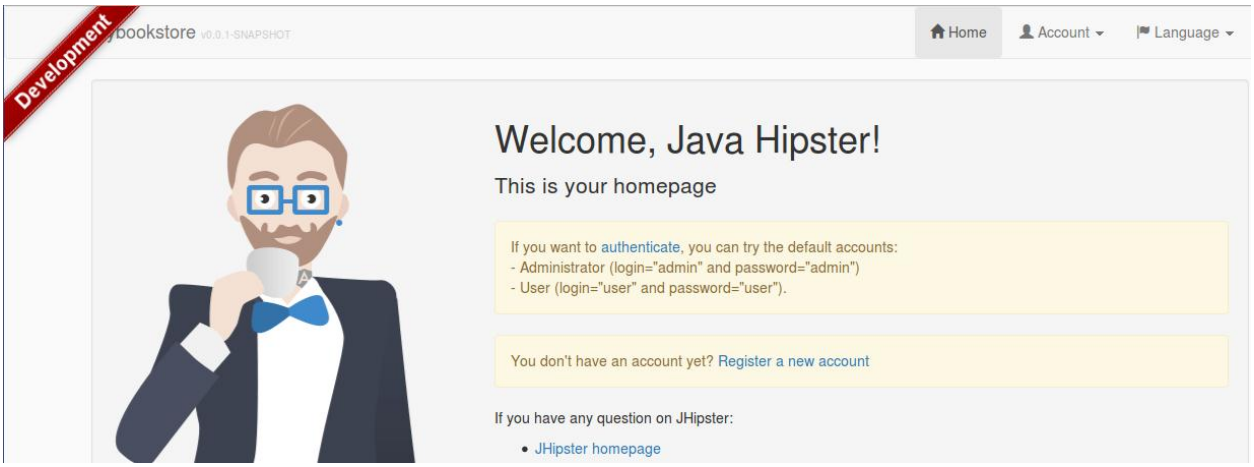


Figure 6: Dev. profile landing page

Server Side Technologies

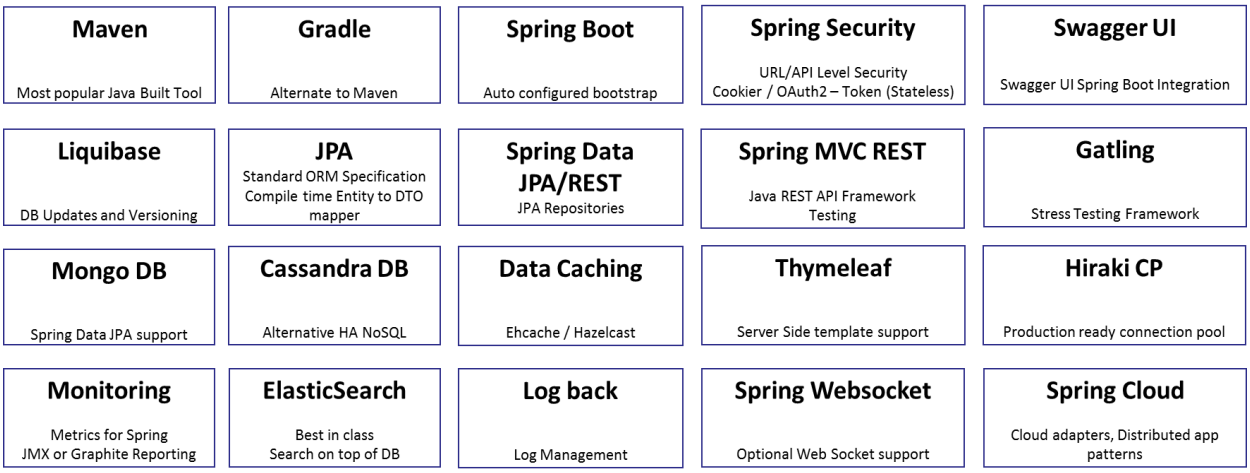


Figure 7: Server side technologies

The server side technology stack core platform is based on Spring and supports both Java 7 and Java 8. There are certain features which are generated only for Java 8, like Elasticsearch, and utilize the lambda functions feature if Java 8 is enabled.

One of the key aspects of designing such a solution is to ensure that security is not an afterthought and takes care of all layers including the UI. There is a known issue with respect to CSRF protection.

Cross Site Request Forgery (CSRF) is a type of malicious attack that occurs when an end-user is forced to execute an unwanted action with or without his/her knowledge, on behalf of the attacker, with the authenticated end user of the application. For example: an email link sent from the attacker posing as a bank or a commercial website, which will perform a function within the website where the end-user is authenticated.

Spring Security has a built-in CSRF protection, which ensures that end-users' data requests are supported by X-CSRF cookie token, which is set in the first GET request and subsequent requests are validated against it. Angular JS too has a CSRF protection built in, however the token format expected is different. The jHipster generated project takes care of this impedance and ensures end-to-end CSRF protection.

Other key security measures include:

- Cookie Theft protection, which ensures that cookies are refreshed for each user login, thus making sure that the older session cookies are unusable by attackers.
- Spring Security token based remember-me support.
- Multiple authentication mechanisms, including stateful cookie based and OAuth2
- Role based access is also integrated with the generated UI, as well as API level access via Spring security annotations.

As mentioned earlier, JHipster is continuously improving and has a very active community. One of the recent additions is the MapStruct which helps in the generation of DTOs for the entities; but this works only with Java 8.

Spring Data JPA

This is a part of the umbrella project of Spring Data, which provides a consistent way of accessing data repositories such as NoSQL databases, and cloud based data services apart from RDBMS. Spring Data JPA provides the JPA repositories implementation and significantly reduces the boilerplate code that has to be written for implementing the DAO layer. The developer can focus on just adding any required queries and utilizing the 'out of the box' features of transaction management, auditing, validation, pagination, etc.

Spring Data REST

Spring Data REST will help to export the Spring Data Repositories as hypermedia based RESTful services, and supports a large number of features for discovering and search entity services.

Swagger UI

The generated jHipster project is integrated with Swagger and all the REST APIs exposed by the project are available as part of the Swagger UI integration, which provides not only an HTML5 compatible API documentation interface, but also acts as a sandbox environment to test out your APIs.

```
@Configuration
@EnableSwagger
@Profile("!" + Constants.SPRING_PROFILE_FAST)
public class SwaggerConfiguration implements EnvironmentAware {

    private final Logger log = LoggerFactory.getLogger(SwaggerConfiguration.class);

    public static final String DEFAULT_INCLUDE_PATTERN = "/api/.*";

    private RelaxedPropertyResolver propertyResolver;

    @Override
    public void setEnvironment(Environment environment) {
        this.propertyResolver = new RelaxedPropertyResolver(environment, "swagger.");
    }

    /**
     * Swagger Spring MVC configuration.
     */
    @Bean
    public SwaggerSpringMvcPlugin swaggerSpringMvcPlugin(SpringSwaggerConfig springSwaggerConfig) {
        log.debug("Starting Swagger");
        StopWatch watch = new StopWatch();
        watch.start();
        SwaggerSpringMvcPlugin swaggerSpringMvcPlugin = new SwaggerSpringMvcPlugin(springSwaggerConfig)
            .apiInfo(apiInfo())
            .genericModelSubstitutes(ResponseEntity.class)
            .includePatterns(DEFAULT_INCLUDE_PATTERN);

        swaggerSpringMvcPlugin.build();
        watch.stop();
        log.debug("Started Swagger in {} ms", watch.getTotalTimeMillis());
        return swaggerSpringMvcPlugin;
```

Figure 8: Spring MVC - Swagger Plugin integration

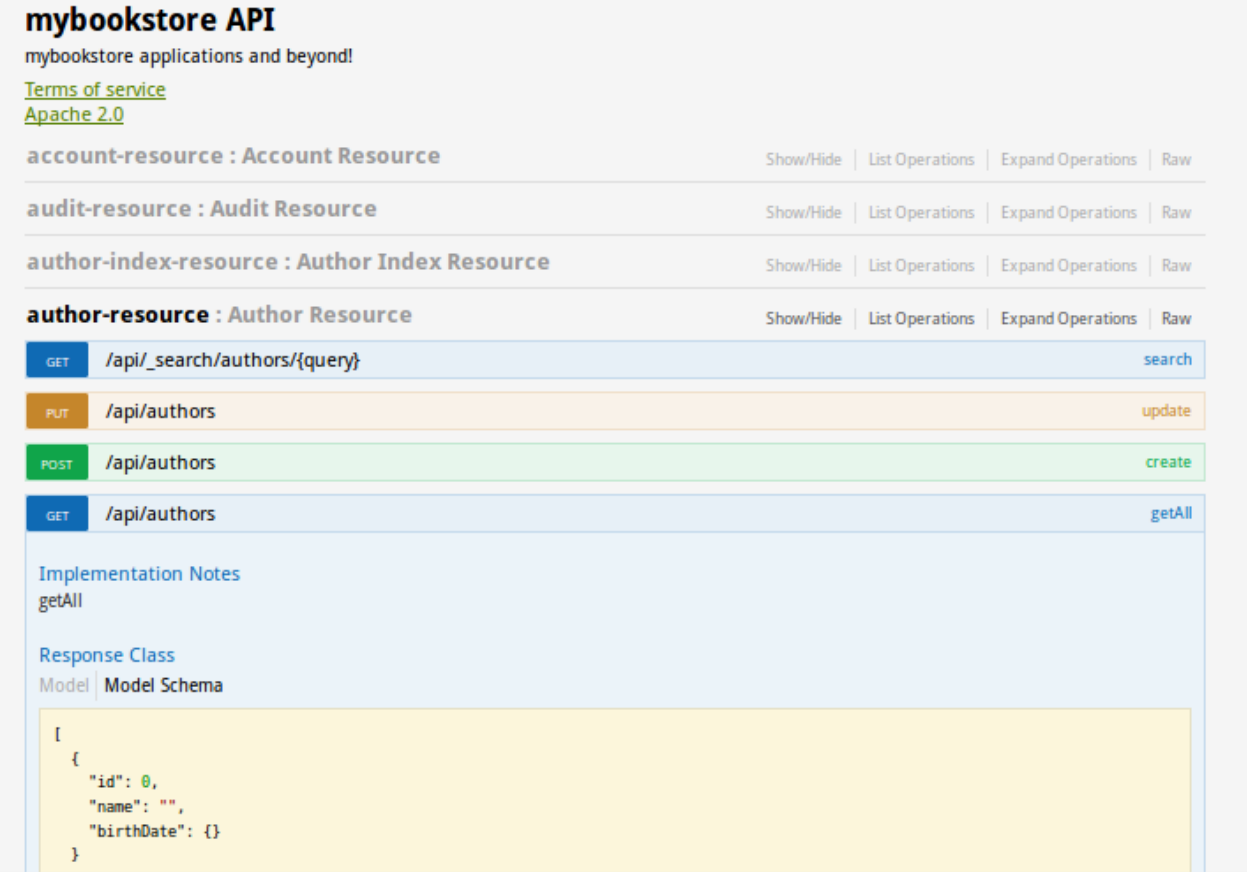


Figure 9: Swagger UI integration

Spring Boot

One of the core components of the jHipster generated project is that it helps in building production grade, self-contained, standalone applications or services. Some of the other key features include:

- Support for metrics, health checks, and externalized configuration, making it production ready
- Intelligent auto-configuration of Spring wherever possible, with no-code generation and zero xml configuration
- Starters or packaged dependencies, which are dramatically simplified for your state-of-the-art, Java and Server development


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
```

Figure 10: Sample Spring Boot Starters added to the generated project

Spring Boot Actuator

Application health metrics is a critical aspect of any distributed web application and integration with Spring Boot Actuator in the generated project automatically exposing two endpoints for metrics collection:

1) /health – to check if a service is up or not
2) /metrics – to read the application metrics

There is an 'out of the box' integration provided with Graphite, which provides a very convenient administrative interface.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator</artifactId>
</dependency>
```

Figure 11: Spring Boot Actuator integration

Apart from the default metrics and health checks that are provided, this integration can easily be extended to provide additional custom metrics that you would want to track as part of the application.

Figure 12: Health API UI screen

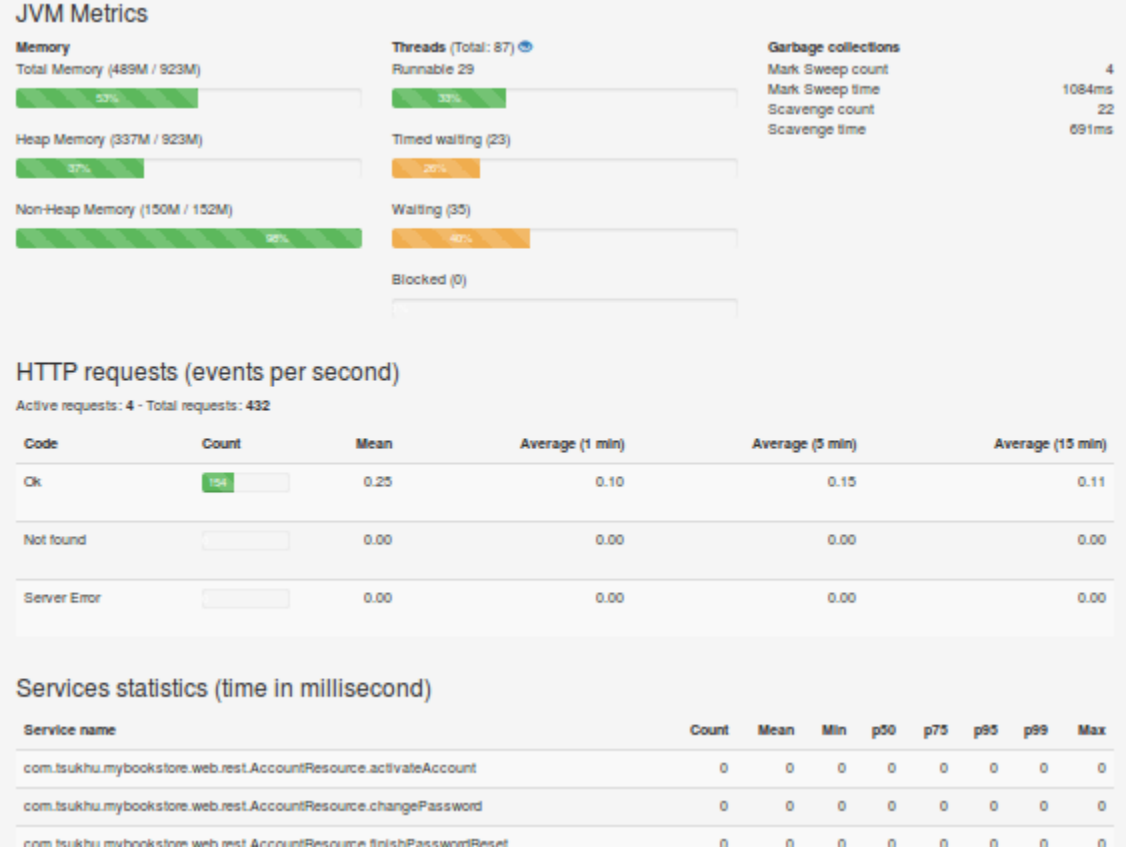


Figure 13: Metrics API UI screen

Ehcache statistics

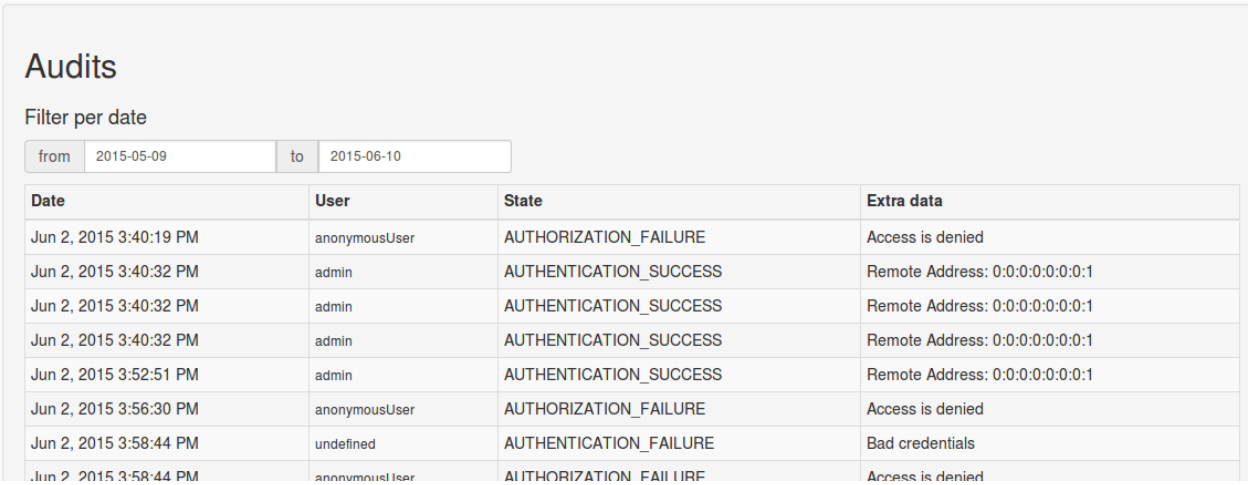
Cache name	Objects	Hits	Misses	Eviction count	Mean get time (ms)
Author	0	0	0	0	
Authority	2	11	0	0	
Book	0	0	0	0	
PersistentToken	0	0	5	0	
User	2	2	0	0	

DataSource statistics (time in millisecond)

Usage (3 / 10)	Count	Mean	Min	p50	p75	p95	p99	Max
30%	48	35.11	0.00	17.00	38.00	63.00	506.00	2,916.00

Figure 14: Metrics API UI screen (cont.).

Spring Boot Actuator Audit Framework is also integrated in the project, which helps in publishing Spring security events like 'authentication success' or 'failure'.



Audits

Filter per date

from 2015-05-09 to 2015-06-10

Date	User	State	Extra data
Jun 2, 2015 3:40:19 PM	anonymousUser	AUTHORIZATION_FAILURE	Access is denied
Jun 2, 2015 3:40:32 PM	admin	AUTHENTICATION_SUCCESS	Remote Address: 0:0:0:0:0:0:0:1
Jun 2, 2015 3:40:32 PM	admin	AUTHENTICATION_SUCCESS	Remote Address: 0:0:0:0:0:0:0:1
Jun 2, 2015 3:40:32 PM	admin	AUTHENTICATION_SUCCESS	Remote Address: 0:0:0:0:0:0:0:1
Jun 2, 2015 3:52:51 PM	admin	AUTHENTICATION_SUCCESS	Remote Address: 0:0:0:0:0:0:0:1
Jun 2, 2015 3:56:30 PM	anonymousUser	AUTHORIZATION_FAILURE	Access is denied
Jun 2, 2015 3:58:44 PM	undefined	AUTHENTICATION_FAILURE	Bad credentials
Jun 2, 2015 3:58:44 PM	anonymousUser	AUTHORIZATION_FAILURE	Access is denied

Figure 15: Audit Log UI Screen

Logging

Figure 16: Log UI screen with log level change functionality

```
@RequestMapping(value = "/logs",
        method = RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
@Timed
public void changeLevel(@RequestBody LoggerDTO jsonLogger) {
    LoggerContext context = (LoggerContext) LoggerFactory.getILoggerFactory();
    context.getLogger(jsonLogger.getName()).setLevel(Level.valueOf(jsonLogger.getLevel()));
}
```

Figure 17: Log API PUT method

Liquibase

Liquibase integration is easily extendable to bootstrap .csv data into data entity tables. In the sample application created, the author and book tables were loaded with .csv data by extending the Liquibase configuration to include a changeset for the data upload.

```
<changeSet id="20150000000000" author="jhipster">
    <loadData encoding="UTF-8"
              file="config/liquibase/authors.csv"
              separator=";"
              tableName="AUTHOR"/>

    <loadData encoding="UTF-8"
              file="config/liquibase/books.csv"
              separator=";"
              tableName="BOOK"/>
</changeSet>
```

Figure 18: Liquibase bootstrapping

Since the Elasticsearch index creation in the project generator is bound to the CRUD REST APIs of the entities, these .csv based bootstrap entries are not picked up. However, you can easily expose a secure admin API to re-index the entities, which are automatically available then in the Swagger UI administrative interface as well.

Spring Cloud

Spring Cloud is a toolbox for building distributed Cloud applications and Microservices using Spring. It brings together a set of design patterns and use cases you will often encounter in building such systems. These include the modules below:

- Configuration Server – Ability to externalize and distribute configuration settings
- Configuration Server Client – Bring server configuration in Spring context
- Integrate with NetFlix Open Source projects
  - Eureka – Service Discovery (Registry and Lookup)
  - Fiegn – Http Client framework
  - Hystrix – for resilience and monitoring (Provides a Circuit Breaker pattern implementation)
  - Ribbon – Inter process communication
  - Zuul – Edge Service (Can be used to implement the API Gateway pattern)
  - Turbine – SSE Stream Aggregator
- AWS adapters for Spring Beans
- Spring Cloud Bus – Light weight message broker
- Connectors – They enable APIs to connect and bind to services provided by Cloud infrastructure providers like Cloudfoundry, Heroku, and Openshift. This enables developers to build their applications to not only run locally, but to also automatically bind to vendor-specific services like databases, without any issues.
- For a detailed list, please refer to Spring io documentation on Spring cloud.

From the jHipster perspective, the configuration server client and cloud connectors are pre-integrated into the generated application, which lets you deploy seamlessly to a cloud environment like Heroku, with minimal effort.

```
<!-- Spring Cloud -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-cloudfoundry-connector</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-spring-service-connector</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-localconfig-connector</artifactId>
</dependency>
```

Figure 19: Spring Cloud dependencies

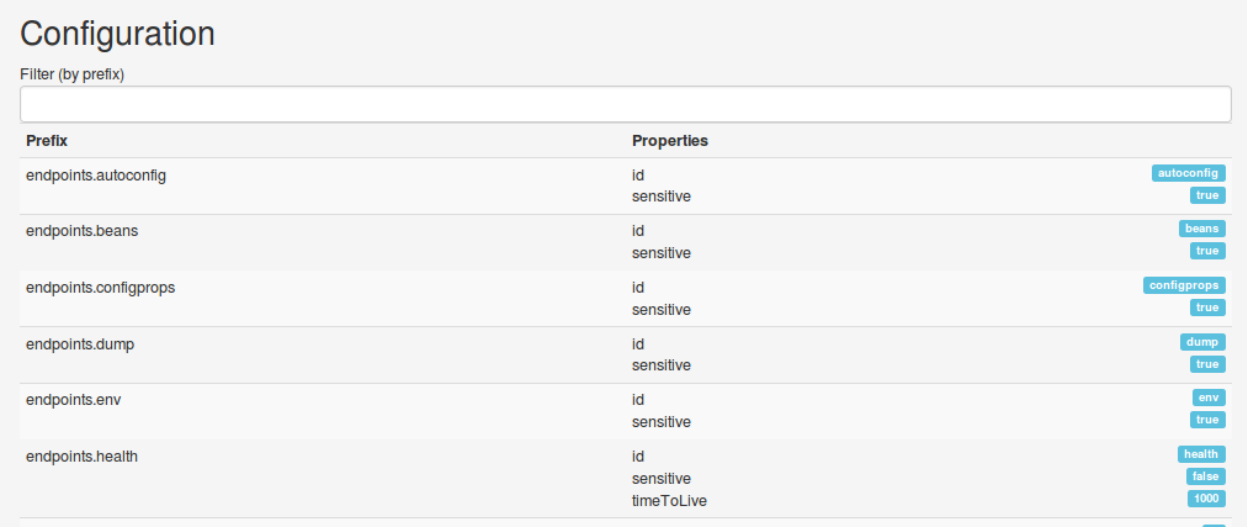


Figure 20: Spring Cloud Configuration UI

For deploying to Cloud environments, there are sub-generators available for Cloud vendors based on Spring Cloud, which help you bind to the services of the Cloud provider as well as push your application to the Cloud.

Mail

Mail integration uses Spring Boot starter for mail and wires up a *JavaMailSender* configuration bean for you. All that is required is to set the email server configuration in the *application.yml* file.

For testing purposes, there is a very useful tool which provides you with the ability to use a dummy SMTP server called mailtrap.io. Here, a free service plan was used to test the user activation and password reset functionality, which comes built-in.

The email server configuration in *application.yml* would look something like this:

```
mail:
    host: mailtrap.io
    port: 2525
    username: [redacted]
    password: [redacted]
    protocol: smtp
    tls: false
    auth: false
    from: mybookstore@localhost
```

Figure 21: Application.yml mail configuration

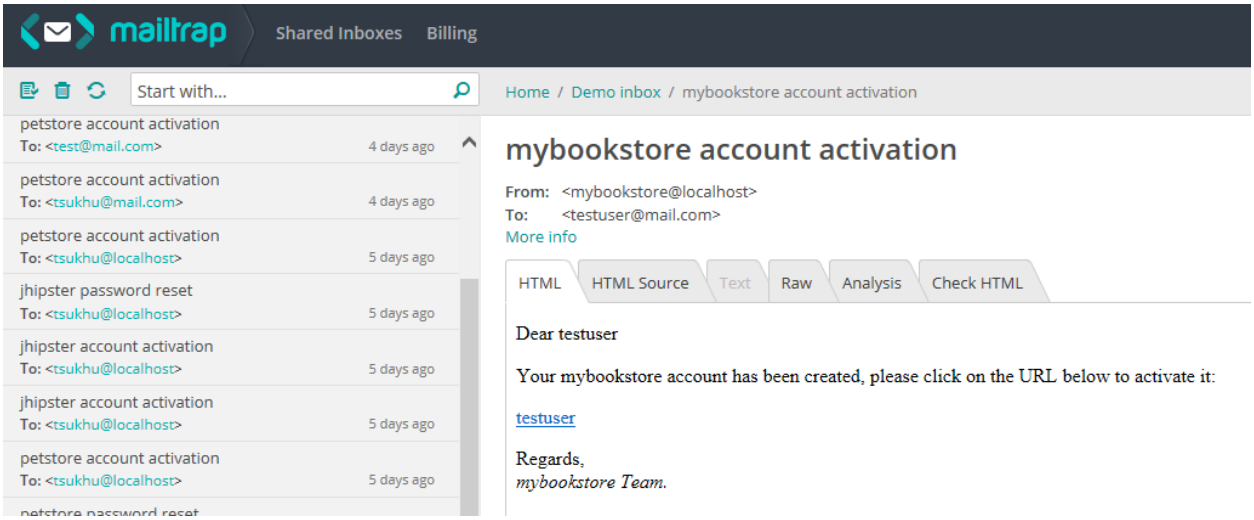


Figure 22: mailtrap.io Dummy STMP Server

Elasticsearch

Elasticsearch support is via a Spring Data Elasticsearch starter module, which exposes Elasticsearch as a Data Repository. The fact that Java 8 is required by jHipster for Elasticsearch support is due to the fact that they use Java 8 Streams as part of the implementation.

One of the new features of Java 8 are Stream operations, which operate on a source data structure (collections / arrays), and produce pipelined data that can be operated upon.

```
/**
 * SEARCH  /_search/authors/:query -> search for the author corresponding
 * to the query.
 */
@RequestMapping(value = "/_search/authors/{query}",
    method = RequestMethod.GET,
    produces = MediaType.APPLICATION_JSON_VALUE)
@Timed
public List<Author> search(@PathVariable String query) {
    return StreamSupport
        .stream(authorSearchRepository.search(queryString(query)).spliterator(), false)
        .collect(Collectors.toList());
}
```

Figure 23: Java 8 StreamSupport APIs for Elasticsearch implementation

Given above is a snippet from the Elasticsearch REST API, where StreamSupport is used to collect the data from the author search repository and return a List.

Performance Testing

Performance testing integration into a project is something we generally delay to a later stage of the project, and start working on this once we have the basic build infrastructure in place. jHipster helps us here by generating a Gatling integration as part of the project. There are test cases automatically added for the entities you add using the jHipster entity generation workflow.

Gatling is an open source stress testing framework built using Scala, Akka, and Netty. The key features being that it is built for high performance stress tests and come with extremely powerful visualization based on HighCharts.

Figure 24: Sample output from Author API Test

The most common stress and load testing tool we are familiar with is JMeter. Apparently, a benchmark exercise done by flood.io (which is a distributed load testing SaaS platform that supports both Gatling and JMeter) concluded that for up to 10K users, there is not much to choose between the two; but, the true value of Gatling comes across for extremely high number of users ~20K where it performs better than JMeter.

High Productivity Setup

One of the key productivity gains with jHipster is the seamless setup of the database and Elasticsearch environments for development and production.

- If you use an SQL database, JHipster will launch an in-memory H2 instance in order to use a temporary database for its integration tests. Liquibase will be run automatically and will generate the database schema.
- If you use Cassandra, JHipster will launch an in-memory Cassandra instance using CassandraUnit.
- If you use MongoDB, as this is not a Java-based database, you will need to run a specific MongoDB instance on your local machine.
- If you use Elasticsearch, JHipster will launch an in-memory Elasticsearch instance using Spring Data Elasticsearch.
- It also provides the option of a quick launch of the spring boot application in dev mode, bypassing the Liquibase, Swagger, and admin services using UnderTow instead of Tomcat. The launch is executed in under 10 seconds (in fact the site mentions 4-6 seconds). This is a high productivity option of developers wanting to focus on rolling out a new feature quickly.

Conclusion

Though the jHipster project is just about 18 months old, it has an ever growing list of contributors and community members. The choice of technology stacks, with seamless integration, and the detailed level of features that get packaged in the generated project, make it a very compelling option for building next generation distributed web applications. The ability to have all this, setup in a matter of a few generation workflow steps, helps reduce the turnaround time to produce production-ready web applications. Though the complete application is packaged into one generated project, this can easily be split into different service layers, to form the base of a Microservices based architecture.

References

- JHipster Site http://jhipster.github.io
- Spring Boot http://projects.spring.io/spring-boot/
- Spring Data http://projects.spring.io/spring-data/
- Spring Cloud http://cloud.spring.io/
- Mail Trap https://mailtrap.io/

Author Info

Tarun Kumar Sukhu

Tarun is Deputy General Manager at HCL and has over 19 years of experience in Product Engineering and Consultancy Services, dealing with Data Management Platforms, Cloud, Platform Migration, and Digital e-Commerce.

Hello, I'm from HCL's Engineering and R&D Services. We enable technology led organizations to go to market with innovative products and solutions. We partner with our customers in building world class products and creating associated solution delivery ecosystems to help bring market leadership. We develop engineering products, solutions and platforms across Aerospace and Defense, Automotive, Consumer Electronics, Software, Online, Industrial Manufacturing, Medical Devices, Networking & Telecom, Office Automation, Semiconductor and Servers & Storage for our customers.

For more details contact**: ers.info@hcl.com**
Follow us on Twitter: http://twitter.com/hclers & LinkedIn: http://lnkd.in/bt8hDXM
View our blog-site: http://www.hcltech.com/blogs/engineering-and-rd-services
Visit our website: http://www.hcltech.com/engineering-rd-services

HCL