



El documental dinámico.

Cómo generar documentación en un paisaje de microservicios

#Java Magazin (<https://jaxenter.de/tag/java-magazin>) #Microservicios (<https://jaxenter.de/tag/microservices>)

 6 de enero de 2020  Simone Görner, Richard Vogel

La buena documentación se considera la figura decorativa de un sistema. Pero sobre todo se considera un mal necesario y, por lo tanto, se descuida con demasiada frecuencia. La documentación incompleta y desactualizada es un área de riesgo con un gran potencial de conflicto y error. Para evitar los riesgos, comenzamos a buscar una solución dinámica y flexible. Te presentamos los resultados aquí.



© Shutterstock / StockVector

La documentación requiere mucho mantenimiento y, a menudo, no corresponde a la realidad, es decir, el código. La tendencia actual de construir sobre microservicios en el entorno empresarial no simplifica el problema: el usuario final no quiere y no debe saber que el sistema está dividido en diferentes subsistemas. Sin embargo, el manual, por lo tanto, sigue siendo monolítico en este punto. La documentación central y uniforme en diferentes sistemas, lenguajes de programación y fuentes es un desafío.

En nuestra empresa encontramos exactamente este problema. Una suite de software que consta de varios subsistemas, algunos de ellos modulares y expandibles específicos del cliente, requiere un manual estandarizado externamente. Dado que las compañías competidoras usan este software directamente, la documentación del cliente A al cliente B no debe pasar bajo ninguna circunstancia por error. Sin embargo, el documental siempre debe estar completo y actualizado. Y, por supuesto, no debería generar ningún esfuerzo.

Lenguaje uniforme

La documentación proviene de una amplia variedad de sistemas fuente. Estos están escritos en diferentes idiomas y proporcionan información en varias estructuras. Para obtener un documento central, primero tenemos que elegir un idioma uniforme para la documentación.

Los desarrolladores aman Markdown (al menos la mayoría). A más tardar, dado que GitHub muestra cada repositorio formateado *adecuadamente* con un *archivo readme.md*, el lenguaje de marcado debe ser parte del repertorio estándar de cada desarrollador. Con el programa de código abierto Pandoc (<https://pandoc.org>), Markdown se puede convertir fácilmente en varios formatos, como HTML o PDF, pero también en EPUB, docx o en un moderno set de filmación Reveal.js.

A continuación, nos centraremos en la creación de un PDF, ya que este formato se usa a menudo para la documentación del software empresarial. Si un archivo de reducción se convierte a PDF usando Pandoc, LaTeX se usa como formato intermedio. Por lo tanto, no es sorprendente que todas las características relevantes que se requieren para una buena documentación sean cumplidas implícitamente por Pandoc.

- Encabezados de diferentes tamaños.
- Formato de texto en negrita, cursiva, subrayado
- Hipervínculos con títulos
- Liza

- Referencias / enlaces dentro del texto
- Notas al pie
- Tablas y estilos de tabla
- Imágenes con subtítulos
- Codifique tanto en línea como en bloque
- Referencias a la literatura, también con diferentes formatos CSL
- Fórmulas matemáticas, tanto en línea como en bloques.
- Bibliografía, ilustraciones, tablas de contenido.
- Referencias a literatura incluyendo un directorio
- Diseñando con tus propias plantillas

Se puede encontrar una reducción de muestra, que contiene la mayoría de las características enumeradas, en el Listado 1. Para convertirlo en un documento PDF, Pandoc no necesariamente tiene que instalarse de forma nativa, hay numerosas imágenes de Docker que traen este programa junto con LaTeX . La *llamada de* línea de comando en el Listado 2 usa la imagen oficial *pandoc/latex* en la versión 2.6. El archivo de *rebajas* referenciado *input.md* debe estar en el directorio actual. El formato de salida del archivo *result.pdf* está determinado implícitamente por la extensión del archivo. El ejemplo de código completo se puede encontrar como una esencia en GitHub (<https://bit.ly/2YPh9ny>) . La llamada correspondiente a la línea de comandos de Windows también se puede encontrar allí.

Listado 1

```
1  ---
2do title: Dynamische Dokumentation
3ro author: Simone Görner und Richard Vogel
4to lang: de
5 5  lof: yes
6 6  ---
7 7
8vo # Pandoc
9 9  ![Pandoc Logo](logo.png){width=15%}
10mo
11 ## Kurzvorstellung
12mo [Pandoc](https://pandoc.org) bietet dicke und schräge Features wi
13
14 [^note]: mehr siehe @book1.
15
dieciséis ```java
17 // Codeblöcke
18vo System.out.println("Mit Highlighting")
Diecinueve ```
20
21 Da Markdown über $LaTeX$ zu PDF transformiert wird, bekommt man Feature
22 $$
23 e^{i\pi}=-1
24 $$
25 geschenkt. Hierzu gehören auch Tabellen und Abbildungen sowie
26
27 - Inhaltsverzeichnis
28 - Abbildungsverzeichnis.
29
30 ## Literaturverzeichnis
```

Listado 2

```
1  # Erstellung eines PDFs aus einer Markdown-Datei samt Referenzen im Bibtex-Format
2do docker run -v `pwd`: /data \
3ro   pandoc/latex:2.6 \
4to   input.md \
5 5   -F pandoc-citeproc \
6 6   --bibliography references.bib \
7 7   --toc \
8vo   -o result.pdf
```

Pandoc es expandible. Entonces, ya hay algunos filtros que p. B. formatear fórmulas químicas o generar diagramas UML a partir de fragmentos de código. Para sus propios requisitos pequeños, el lenguaje de script Lua se puede utilizar para escribir rápida y fácilmente su propio filtro Pandoc.

Dinamizar rebajas?

Casi todos los desarrolladores están familiarizados con el problema de que la documentación debe mantenerse dos veces. Una vez directamente en el código para que otros desarrolladores conozcan la tarea de la clase o método, y además en un documento separado que se pone a disposición del cliente como documentación técnica. Aquí existe un gran riesgo de que solo se cambie una de las dos posiciones en caso de un cambio y esto conducirá a documentación divergente. Este problema se puede resolver con un enfoque dinámico. En principio, los desarrolladores están más motivados para dedicarse a la documentación si se hace directamente en el código, ya que esto también es útil para su propia visión general.

La versatilidad y la facilidad de mantenimiento de los archivos de descuento son otros argumentos positivos. Dado que la gestión de versiones siempre debe (!) Usarse en proyectos de software, los usuarios pueden comprender directamente los cambios en la documentación, lo que también hace que sea muy fácil de mantener.

El enfoque dinámico generalmente crea una oportunidad para la documentación continua. La automatización está desempeñando un papel cada vez más central; también es deseable para la documentación. En cualquier caso, se siente más cómodo para los desarrolladores "programar" la documentación que escribirla.

Otra ventaja es que la documentación puede ser probada. Por ejemplo, si desea documentar una llamada de línea de comando, puede usar una prueba unitaria para verificar si es correcta. Del mismo modo, se pueden implementar pruebas de humo relativamente simples para garantizar la exactitud de los archivos de texto generados. Por lo tanto, busque la palabra clave "cliente B" en el documento para el cliente A para excluir el peor de los casos.

Con todas las ventajas del enfoque dinámico, no debemos perder de vista las desventajas. Debido a los ajustes continuos en varios puntos, existe el riesgo de que algunas cosas ya no funcionen como antes sin que se noten directamente. Verificar manualmente la documentación generada automáticamente cada vez que se pierda el objetivo. Por lo tanto, si se produce un error, debemos asegurarnos de que no vuelva a aparecer en el futuro. ¿Por qué no simplemente usar una prueba de regresión automatizada?

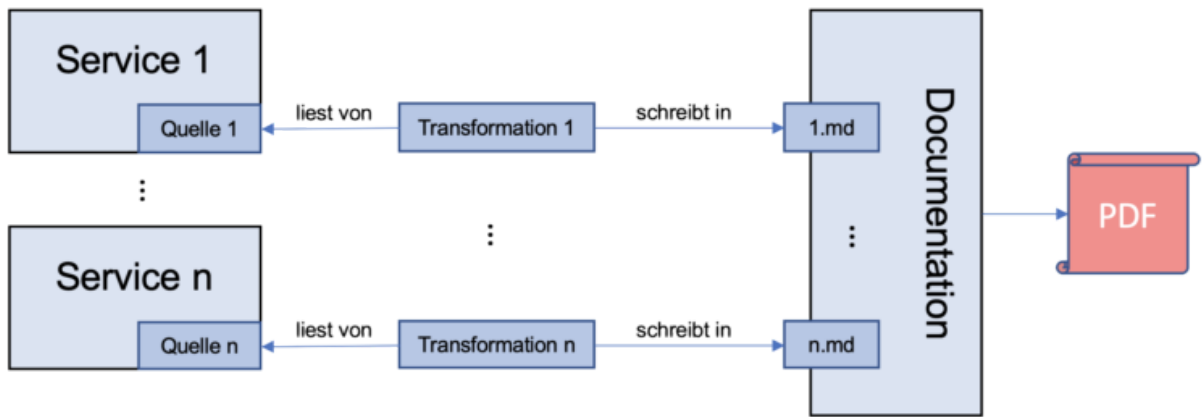
Otro punto que puede conducir a problemas es la creciente complejidad. El know-how requerido aumenta con cada fuente adicional utilizada. Esto hace que sea más difícil para un nuevo colega, por ejemplo, comprender y expandir el constructo, y la documentación debe explicarse por sí misma y no debe documentarse nuevamente.

Además, toda la idea se limita a Markdown y Pandoc. Si uno alcanza los límites del rango de funciones aquí, las extensiones del panorama de herramientas requieren tiempo adicional. Sin entrar en más detalles, queremos referirnos a RMarkdown (<https://rmarkdown.rstudio.com>) , una forma extremadamente poderosa de dinamizar (<https://rmarkdown.rstudio.com>) documentos de reducción y expandirlos con elementos programáticos, matemáticos y de diseño.

¿Quieres modularidad?

En una arquitectura compleja de microservicios, la modularidad se capitaliza, aunque un manual o documento central tiene un carácter más monolítico. Si varios sistemas se expanden individualmente y ciertos microservicios solo se activan para ciertos clientes, se debe reconsiderar el papel de la unidad de creación de documentación.

Si un microservicio ofrece la función X, que está documentada con la fuente de datos Y, la documentación solo debe crearse si este servicio está disponible. Aquí utilizamos el conocido patrón de "Inversión de control", de modo que el módulo de creación de documentación simplemente combina los fragmentos existentes en un documento central. No le importa el contenido concreto, sino la estructura, la plantilla externa y el proceso de creación. Las excepciones confirman la regla: por supuesto, a nadie se le impide posicionar uno u otro texto estático en este módulo. **La figura 1** ilustra este proceso.



(https://jaxenter.de/wp-content/uploads/2019/11/vogel_goerner_doku_1.png)

Fig. 1: Proceso modularizado de creación de documentación.

Los ejemplos dicen más de 1,000 palabras.

Algunos de estos enfoques son abstractos. Para tener una idea de cómo se ve en realidad, hemos recopilado algunas ideas:

- Creación de notas de lanzamiento de rastreadores de problemas como Jira
- Documentación de una herramienta de línea de comandos con comandos de ejemplo.
- Diagramas de infraestructura de una configuración docker-compose.yml
- Guía del usuario con capturas de pantalla y pasos
- Presentación de la documentación Swagger de un servicio REST.
- Descripción general de los resultados de la prueba de la ejecución actual de Jenkins
- Representación de números de rendimiento en diagramas.

Explicaremos los primeros cuatro de estos enfoques a continuación y proporcionaremos ejemplos de código. Estos ejemplos deberían permitirle comprender mejor la implementación y utilizarla para sus propios proyectos.

Ejemplo 1: Notas de la versión

Las notas de la versión son parte de la entrega de un producto de software. Contienen historias implementadas y errores corregidos junto con un breve resumen y se refieren a más información. A muy pocos se les ocurrirá la idea de mantener manualmente las notas de la versión. Cada herramienta de seguimiento de problemas actual incluye una conveniente función de exportación para todos los problemas asignados a ciertas versiones.

Sin embargo, estamos buscando una solución totalmente automatizada para nuestro propósito. El complemento Maven Changes es ideal para trabajar con Maven. La conexión a un rastreador de problemas como Jira o GitHub puede configurarse allí, y otros sistemas pueden conectarse a través de una interfaz. El Listado 3 muestra un ejemplo de configuración Maven de este complemento. Los parámetros dependientes de Jira, como las ID de estado y los datos de acceso, por supuesto, deben ajustarse según el contexto. La activación se lleva a cabo explícitamente llamando a `mvn cambios:anuncio-generar -DVERSIÓN=1.2.3` o implícitamente z. B. dentro de la fase del paquete. Un ejemplo completo se puede encontrar como una esencia (<https://bit.ly/2OWoaOA>) .

Listado 3

```

1  ...
2do <issueManagement>
3ro   <system>Jira</system>
4to   <url>https://jira.url/browse/MyProject</url>
5 5   </issueManagement>
6 6
7 7   <plugins>
8vo   <plugin>
9 9     <groupId>org.apache.maven.plugins</groupId>
10mo   <artifactId>maven-changes-plugin</artifactId>
11     <version>2.12.1</version>
12mo   <configuration>
13     <version>${VERSION}</version>
14     <issueManagementSystems>
15       <issueManagementSystem>JIRA</issueManagementSystem>
dieciséis </issueManagementSystems>
17     <webUser>${jira.user}</webUser>
18vo     <webPassword>${jira.password}</webPassword>
Diecinueve <template>template.vm</template>
20     <templateEncoding>UTF-8</templateEncoding>
21     <announcementFile>ReleaseNotes.md</announcementFile>
22     <resolutionIds>1,6</resolutionIds>
23     <statusIds>6</statusIds>
24     <filter>project = "MyProject"
25       AND fixVersion = ${VERSION}
26       AND issuetype in (Story, Bug)
27       AND status = 6
28       AND resolution in (1, 6)
29     </filter>
30   </configuration>
31 </plugin>
32 </plugins>
33  ...

```

Por defecto, los problemas se extraen en un formato XML uniforme. Sin embargo, el complemento ofrece la opción de convertir los problemas en un formulario definido por el usuario utilizando una plantilla de velocidad. El resultado de la generación resulta en un archivo de marcado legible que se puede agregar a nuestro módulo de documentación.

Los problemas están disponibles aquí como texto sin formato. En realidad, el cliente solicita que los números se vinculen al rastreador de problemas para mayor comodidad. Aquí podemos demostrar el uso simple de un filtro Pandoc. Dicho filtro recorre cada línea de un documento de reducción y, después de una expresión regular, decide si una palabra debe reemplazarse por un enlace. Dado que el rastreador de problemas está solucionado, codificamos el patrón de URL aquí. El Listado 4 *da* una impresión de un filtro Lua. *Para* activarlo, simplemente *agregue* `-lua-filter = <filtername>.lua` al comando.

Listado 4

```

1  local url = "https://jira.url/browse/"
2do
3ro  function Str(el)
4to    if (el.text ~= nil) and (string.find(el.text, "MYPROJECT%-%d+")) then
5 5      story = string.match(el.text, "MYPROJECT%-%d+")
6 6      _url = url .. story
7 7      _title = "Link zur Story " .. story
8vo      return pandoc.Link({pandoc.Str(story)}, _url, _title)
9 9    end
10mo  end

```

Ejemplo 2: cliente de línea de comando

Por lo general, ofrecemos una herramienta de línea de comandos para nuestros productos para que los procesos de procesamiento se puedan llamar automáticamente, p. B. por trabajos cron externos. Aquí es molesto si la llamada de línea de comando específica no coincide con la documentación especificada. Con el mantenimiento manual de la documentación, la corrección es difícil de verificar automáticamente: hablamos por experiencia.

Si estamos en un proyecto Java, podemos abordarlo con pragmatismo. El requisito previo es que los comandos individuales sigan una interfaz que proporcione el nombre, la descripción y los parámetros u opciones. Ahora todas las instancias de esta interfaz se pueden resolver mediante reflexión y se puede generar un código de descuento. No es un enfoque agradable y reutilizable, pero efectivo.

La biblioteca de código abierto JCommander nos libera de algunas tareas desagradables al crear una herramienta de línea de comandos. Analizar y validar parámetros de entrada, pero también generar una página de ayuda son parte del rango de funciones. Afortunadamente, la estructura específica de esta página de ayuda es intercambiable y se delega a las implementaciones de la interfaz IUsageFormatter. Simplemente entregue el *MarkdownFormatter* escrito a la instancia de *JCommandery*, en lugar de la página de ayuda, obtendremos un archivo de descuento que podemos agregar a nuestra documentación. Un ejemplo de implementación de un *formato de descuento* se muestra en el Listado 5. Introduce el comando de línea de comando del Listado 6 en el formato uniforme del Listado 7El ejemplo completo se puede encontrar en Gist (<https://bit.ly/2Klttff>) .

Si está utilizando un marco de línea de comando diferente, también puede tener suerte. Al igual que JCommander, la CLI de Apache's Commons o el marco emergente picocli también ofrecen la posibilidad de intercambiar la representación de la página de ayuda para nuestros propósitos.

Listado 5

```
1 public class MarkdownUsageFormatter implements IUsageFormatter {
2do ...
3ro public void usage(StringBuilder out, String indent) {
4to   commander.getFields().forEach(
5 5     (k, v) -> out.append("- **")
6 6         .append(k.getName())
7 7         .append("*: ")
8vo        .append(v.getDescription())
9 9         .append("\n"));
10mo out.append("\n\n").append("## Kommandos\n");
11   for (String c : commander.getCommands().keySet()) {
12mo     out.append("\n### Kommando: **").append(c)
13         .append("**\n").append("#### Beschreibung")
14         .append("\n\n");
15   ...
dieciséis out.append("#### Parameter\n\n");
17   ...
18vo   fields.forEach(
Diecinueveavo   (k, v) -> out.append("- **")
20         .append(k.getName())
21         .append("*: ")
22         .append(v.getDescription())
23         .append("\n")
24     );
25   out.append("\n#### Beispielaufruf\n\n")
26       .append("```\n")
27       .append("docker exec client run ")
28       .append(c);
29   ...
30   }
31 }
32 }
```

Listado 6

```
1 @Parameters(commandDescription = "Import der Stammdaten (JSON files).", com
2do public class StammdatenImportJob implements Job {
3ro
4to   @Parameter(names = {"-encoding"}, description = "Encoding der Dateien")
5 5   private String encoding;
6 6
7 7   @Parameter(names = {"-filter"}, description = "Dateinamenfilter")
8vo   private String filter;
9 9
10mo   @Parameter(description = "Verzeichnis der zu importierenden Dateien", ari
11   private Path targetDir;
12mo
13   @Override
14   public void run() throws ClientException {
15       // Konkrete Implementierung
dieciséis   }
17 }
```

Listado 7

```
1  ## Kommandos
2do  ### Kommando: **importStammdaten**
3ro  #### Beschreibung
4to
5 5  Import der Stammdaten (JSON files).
6 6
7 7  #### Parameter
8vo
9 9  **targetDir** (Hauptargument): Verzeichnis der zu importierenden Dateie
10mo
11  - **encoding**: Encoding der Dateien
12mo  - **filter**: Dateinamenfilter
13
14  #### Beispielaufruf
15
dieciséis  ```
17  docker exec client run importStammdaten \
18vo      -encoding <value> \
Diecinueveavo      -filter <value>\
20      <targetDir>
21  ```
```

Ejemplo 3: diagramas UML

Los diagramas de distribución son una de las tareas obligatorias de una arquitectura de software bien documentada. Fiel al lema "Una imagen vale más que mil palabras", este diagrama aumenta enormemente la calidad de la documentación.

Si un equipo de desarrollo se adhiere al paradigma de "Infraestructura como código", a menudo hay un archivo de configuración para un sistema completamente funcional. Si ahora se genera un diagrama a partir de esta configuración, el arquitecto de software guarda continuamente la comparación de la infraestructura y los diagramas UML. Si las infraestructuras difieren de las instalaciones individuales de los clientes, el script de generación se puede reutilizar en varias instalaciones.

PlantUML puede usarse para generar diagramas en formato de texto versionable. Una sintaxis pegadiza crea diagramas UML robustos, que también puede organizar visualmente con un estilo individual. Y gracias a la expansión de Pandoc con filtros, la transformación del código PlantUML puede integrarse en el proceso.

Pero ahora un ejemplo concreto: si el equipo entrega su software en contenedores Docker, la infraestructura completa se puede describir en un archivo *docker-compose.yml*. El entorno de integración continua prueba automáticamente contra esta configuración para garantizar la integridad.

El Listado 8 indica el inicio de dicho archivo de configuración. Un proxy nginx encapsula varias aplicaciones, como el contenedor "webapp", que a su vez requiere que se ejecute el contenedor "webappdb". Dado que cada contenedor se describe con un nombre, una descripción y las dependencias asociadas, este archivo *ym* puede leerse con un lenguaje de script y transformarse como PlantUML como en el Listado 9. Esto proporciona un diagrama claro de un entorno con varios contenedores, como se muestra en la **Figura 2**. Un script de muestra (<https://bit.ly/2Z0PSJV>) aparece en Gist (<https://bit.ly/2Z0PSJV>).

El uso de un archivo de configuración para Kubernetes también es posible como punto de partida. Por supuesto, es importante en este punto que se documente exactamente la infraestructura en la que se realizan las pruebas.

Listado 8

```

1 | version: '3'
2do
3ro  services:
4to   proxy:
5 5   image: nginx:1.15-alpine
6 6   labels:
7 7   description: "nginx SSL Proxy"
8vo   depends_on:
9 9   - webapp
10mo   ...
11   webappdb:
12mo   image: postgres:9.6-alpine
13   labels:
14   description: "Webapp DB"
15   webapp:
dieciséis  image: enowa/webapp:1.0.0
17   labels:
18vo   description: "Webapp"
Diecinueve  depends_on:
20   - webappdb
21   ...
22   ...

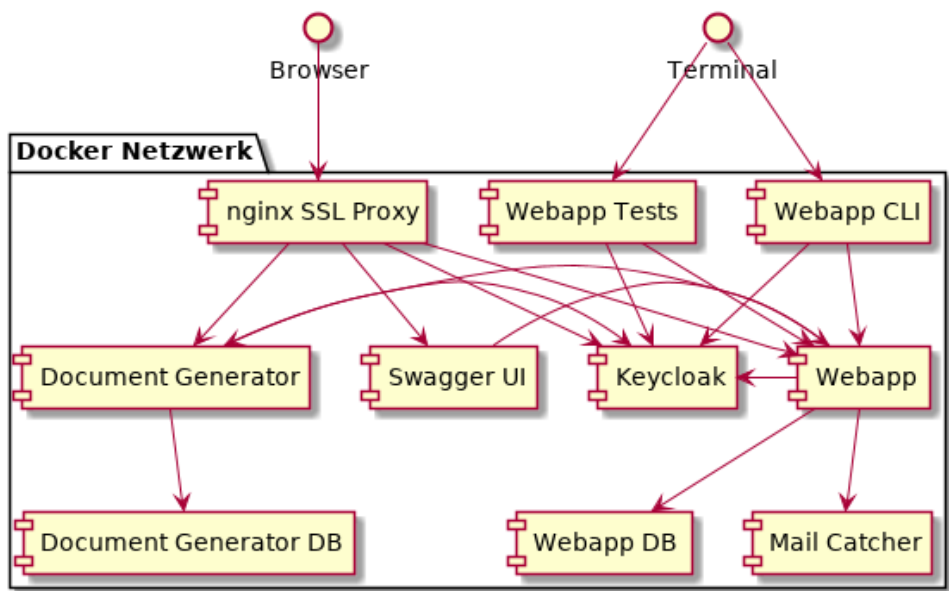
```

Listado 9

```

1 | ## Verteilungssicht
2do
3ro  ``uml
4to  @startuml
5 5  package "Docker Netzwerk" {
6 6  [nginx SSL Proxy] as proxy
7 7  [Webapp DB] as webappdb
8vo  [Webapp] as webapp
9 9  ...
10mo }
11  Browser --> proxy
12mo proxy -down-> webapp
13  webapp --> webappdb
14  ...
15  @enduml
dieciséis ``

```



(https://jaxenter.de/wp-content/uploads/2019/11/vogel_goerner_doku_2.png)
 Fig. 2: Vista de distribución generada automáticamente como un diagrama UML

Ejemplo 4: Guía del usuario

Ahora llegamos a un ejemplo algo más complejo, la documentación de la interfaz de usuario. Aquí tenemos dos factores principales de gastos: en primer lugar, pruebas automatizadas de la GUI, que a menudo se implementan utilizando un navegador controlado a distancia, y en segundo lugar, la documentación del usuario, que generalmente se mantiene manualmente con descripciones y capturas de pantalla. La documentación se vuelve obsoleta no solo cuando algo cambia en los campos de formulario mostrados, sino también indirectamente cuando algo cambia en el diseño externo.

La pirámide de prueba muestra que las pruebas de la interfaz de usuario como pruebas de integración requieren mucho tiempo y, por lo tanto, son muy caras. La reutilización pagaría directamente. Queremos presentar un concepto que combine pruebas automatizadas con la creación del manual del usuario. Esto no solo le garantiza que el manual del usuario corresponde al estado actual, sino que lo hace con la seguridad de una prueba automatizada.

Una prueba de la interfaz de usuario se puede dividir en tres áreas de responsabilidad:

- Suministro de una base de datos
- Secuencia de entradas del usuario
- verificación

Ahora estamos agregando un área de responsabilidad, a saber, la de los fragmentos de documentación. En el Listado 10 mostramos una prueba de ejemplo escrita en Java con Selenium. Con la anotación *autoescrita* `@ImportFile` creamos la base de datos (esto sucede en forma de un archivo Excel interpretado con JexUnit (<https://github.com/fhm84/jexunit>)). Luego usamos el patrón de objeto de página fluido. Este procedimiento tiene dos ventajas: gracias a Autocompletar, escribir una prueba de GUI es fácil, por otro lado, la prueba es legible para todos (¡incluido el cliente!). La implementación concreta basada en selenio permanece oculta para aquellos que técnicamente especifican las pruebas.

Listado 10

```
1 public class SampleIT extends SeleniumTestCase {
2     @Test
3     @Login
4     @ImportFile(name = "brief/Datenbasis.xls")
5     public void testErfolgreicherVersand() throws Exception {
6         new VertragPage()
7             .setLogStrategy(new MarkdownStrategy())
8             .title("handbuch.briefversand.title")
9             .describe("handbuch.briefversand.description")
10            .step("handbuch.vertrag.open.existing")
11            .open("Test-4711")
12            .step("handbuch.vertrag.brief.click.button")
13            .clickSendBrief()
14            .step("handbuch.briefe.select")
15            .selectBrief(4)
dieciséis            .takeScreenshot("handbuch.briefe.dialog")
17            .step("handbuch.briefe.versandoption.choose")
18            .assertDialogVisible(true)
Diecinueve            .chooseVersandOption(Versandoption.LOKALER_DRUCK)
20            .clickButtonCreate()
21            .step("handbuch.briefe.button.confirm")
22            .assertPage(VertragPage.class)
23            .step("handbuch.briefe.download.confirm")
24            .assertFileDownloadExists();
25    }
26 }
```

En el ejemplo, la prueba de GUI envía una carta para un contrato de seguro. Además de las acciones técnicas como *abrir* y verificaciones como *afirmarDialogVisible*, existen métodos para la documentación. En el *método* `logTitle` o `logDescription`, se registran el encabezado y la descripción de la aplicación técnica. Siguen varios pasos clave de procesamiento, que se registran en el método de *pasos*. En el medio, se crea cualquier cantidad de capturas de pantalla, que luego se adjuntan a la documentación. En la **Figura 3** da una impresión de un documento listo generado que incluye una captura de pantalla. Una esencia completa (<https://bit.ly/2MmtsB9>) se puede encontrar en Github.

La forma en que se maneja esta información depende únicamente de la implementación específica de la estrategia de registro. En el ejemplo, no se utiliza texto continuo, sino referencias a un paquete de mensajes para obtener documentación multilingüe.

Versenden eines Briefes

Um außerhalb der Dunkelverarbeitung einen Brief an eine versicherten Person oder einen Korrespondenzpartner zu versenden, führen Sie bitte folgende Schritte durch:

- Öffnen Sie einen existierenden Vertrag
- Klicken Sie in der Aktionsleiste auf den Button “Freier Brief”
- Wählen Sie den gewünschten Brief aus.
- Wählen Sie im Dialog die Versandoption “Manuelle Nachbearbeitung” aus.
- Klicken Sie auf “Brief erstellen”
- Der Brief wurde erstellt und heruntergeladen.

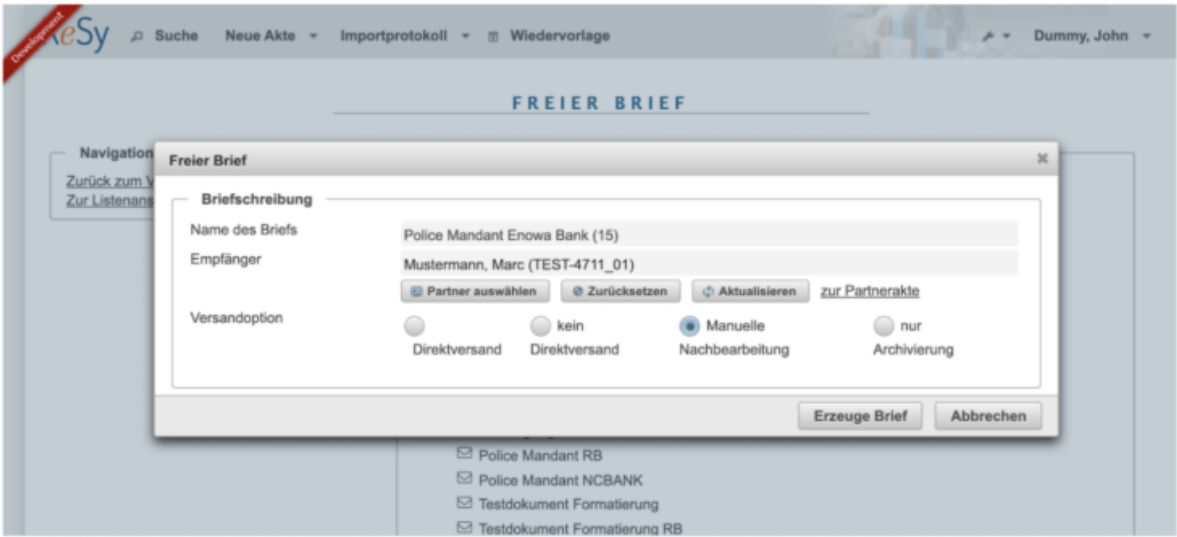


Figure 1: Dialogfenster zum Versand eines Briefes

(https://jaxenter.de/wp-content/uploads/2019/11/vogel_goerner_doku_3.png)

Fig. 3: Manual del usuario generado automáticamente

Conclusión

Nuestros ejemplos ilustran que la automatización puede tener sentido en el lado de la documentación. En última instancia, tenemos que sopesar de un proyecto a otro hasta qué punto el esfuerzo inicial (ciertamente no insignificante) vale la pena. Pero incluso los gastos relativamente altos a menudo se amortizan con el tiempo.

TEMAS RELACIONADOS:

Revista Java (<https://jaxenter.de/tag/java-magazin>)

Microservicios (<https://jaxenter.de/tag/microservices>)

ESCRITO POR



Simone Görner

Simone Görner ha estado trabajando para enowa AG desde 2016 y trata temas relacionados con la automatización de pruebas y el desarrollo de software. Su enfoque principal es el desarrollo posterior del marco de prueba interno y la creación de casos de prueba automatizados.

Todas las contribuciones de Simone Görner
(<https://jaxenter.de/author/simonegoerner>)



Richard Vogel

Richard Vogel ha estado trabajando para enowa AG desde 2015 y trata temas relacionados con el desarrollo de software y la arquitectura. La atención se centra en software expandible modular y específico del cliente, así como en microservicios con Docker.

Todos los mensajes de Richard Vogel
(<https://jaxenter.de/author/richardvogel>)

