TRANSFORMACIÓN DIGITAL DEL ESTADO

TRANSVERSAL

GUÍA TÉCNICA

LINEAMIENTOS PARA EL DESARROLLO DE SOFTWARE

División de Gobierno Digital

Versión 02 | Mayo 2021

gob digital CHILE

Ministerio Secretaría General de la Presidencia

Gobierno de Chile

Índice

I. Introducción

La adopción de buenas prácticas es fundamental para todas las etapas de desarrollo de un sistema o aplicación informática.

Esta guía técnica entrega los lineamientos generales y recomendaciones específicas que deben seguir los equipos que desarrollen software al interior de la Administración del Estado y los equipos de desarrollo de los proveedores, contribuyendo a la construcción de sistemas de alta calidad en todas las instituciones.

La adopción de buenas prácticas es fundamental para todas las etapas de desarrollo de un sistema o aplicación informática. La mitigación de problemas de software permite un uso correcto e íntegro de estos sistemas, ayudando al usuario final a tener un recurso eficiente, confiable, seguro y privado. Asimismo, se minimizan los riesgos y costos de mantención del sistema en horas hombre.

Al desarrollar software, se deben tener en cuenta diversos factores que impactan directa e indirectamente en su correcto uso. Estos factores abarcan desde la comunicación en el equipo de desarrollo y pruebas, pasando por consideraciones dentro de la calidad del código fuente, específicamente factores como seguridad, QA y finalmente el deployment, así como su futuro uso en producción.

Para ello, es fundamental adoptar una metodología de desarrollo de software rápida, flexible e incremental, para minimizar el riesgo de desviaciones en su proceso de creación e implementación, y permitir las adecuaciones tempranas, basadas en la retroalimentación de los usuarios o en las modificaciones de los requisitos durante el proyecto.

Los aspectos que serán abordados en esta guía son:

Calidad del software, tanto en servicio implementado, como en código fuente resultante.

Entregas tempranas en base a productos mínimos viables.

Comunicación efectiva dentro del equipo de desarrollo y también con los stakeholders.

Uso de estándares comunes entre instituciones.

II. Consideraciones iniciales para el desarrollo de aplicaciones

Considerando que el ciclo de desarrollo podría no terminar nunca, se sugiere utilizar todos o algunos de los siguientes ambientes para cada proyecto: producción, staging (demo o certificación), test y development (desarrollo), cada uno de ellos representado en una rama del proyecto en el repositorio de código.

1. Metodologías de desarrollo

Para el análisis, desarrollo e implementación se sugiere utilizar metodologías ágiles, incorporando al menos las recomendaciones de la metodología denominada The Twelve-Factor App, con el objeto de satisfacer, al menos, los siguientes aspectos:

- Usar formatos declarativos para la automatización de la configuración. Así es posible minimizar el tiempo y coste que supone que nuevos desarrolladores se unan al proyecto.
- Tener un contrato claro con el sistema operativo sobre el que se trabaja, ofreciendo la máxima portabilidad entre los diferentes entornos de ejecución.
- Disponer, por defecto, despliegues en plataformas modernas en la nube, obviando la necesidad de servidores y administración de sistemas.
- Minimizar las diferencias entre los entornos de desarrollo y producción, posibilitando un despliegue continuo para conseguir la máxima agilidad.
- Poder escalar la arquitectura o las prácticas de desarrollo sin cambios significativos en términos de herramientas.

Se recomienda al equipo de desarrollo el uso de metodologías ágiles, tales como Scrum o Kanban, Programación Extrema, Modelamiento Ágil, Feature Driven Development (FDD) o cualquier otra que considere relevante; o bien otras metodologías de desarrollo de productos como Shape Up.

Dado que actualmente las instituciones pueden no contar con un equipo con los conocimientos de metodologías de desarrollo ágil u otras, se recomienda

iniciar las capacitación para trabajos futuros y, de esta forma, iniciar los nuevos proyectos con estas metodologías y, con el tiempo, adquirir la experiencia para aplicarla a proyectos ya existentes.

2. Ambientes de desarrollo

Considerando que el ciclo de desarrollo podría no terminar nunca, se sugiere utilizar todos o algunos de los siguientes ambientes para cada proyecto: producción, staging (demo o certificación), test y development (desarrollo), cada uno de ellos representado en una rama del proyecto en el repositorio de código. Y de esta forma, poder detectar rápidamente cuál código es el que está en producción. También se sugiere usar tags en el repositorio para marcar los releases.

Al contar con los resguardos descritos en esta guía, cada paso de un ambiente a otro debiera propender a contar con procedimientos de calidad de código, para obtener un código revisado y apto para su entrega en el ambiente en el que es requerido.

III. Consideraciones para un desarrollo seguro

Las librerías y frameworks de terceros confiables que incorporan mecanismos de seguridad son fundamentales para evitar errores de implementación en áreas en las que el desarrollador no se encuentre tan familiarizado.

1. Uso de librerías y frameworks de seguridad

Se deben seleccionar librerías y frameworks de autores confiables, con mantención y desarrollo activo, y que sean ampliamente utilizadas (y por ende, validadas).

Las librerías y frameworks de terceros confiables que incorporan mecanismos de seguridad son fundamentales para evitar errores de implementación en áreas en las que el desarrollador no se encuentre tan familiarizado.

Es importante inventariar, catalogar y versionar dichas librerías y frameworks para mantenerlas actualizadas y prevenir vulnerabilidades en ellas.

Otra alternativa recomendada es encapsular la librería y exponer sólo la funcionalidad requerida en el sistema que se está desarrollando.

Algunas librerías y frameworks recomendados se encuentran en la sección Tecnologías de Preferencia.

2. Consultas seguras a las bases de datos

Las vulnerabilidades de tipo SQL injection ocurren cuando datos no confiables son incorporados de forma dinámica a una consulta SQL (muchas veces a través de la concatenación directa de dos strings). Este tipo de ataques permite a un adversario extraer, modificar o eliminar datos desde la base de datos e, incluso en ocasiones, tomar total control del sistema comprometido.

Para mitigar o prevenir este tipo de ataques, se deben implementar medidas de protección acordes a la tecnología y plataforma utilizada:

En el caso de estar utilizando un lenguaje orientado a objetos, se sugiere el uso de ORM debidamente probados

por la comunidad e industria . En el caso de utilizar un ORM, se debe asegurar que la herramienta escogida no tenga vulnerabilidades de inyección, e implementar otros mecanismos de defensa como codificar y escapar los datos, como se explica en la próxima sección de esta guía. Adicionalmente, se sugiere validar que el ORM escogido responda adecuadamente ante altas cantidades de transacciones.

En el caso de no utilizar ORM, se deben utilizar prepared statements para prevenir posibles inyecciones de código SQL.

En el caso de contar con tecnología de base de datos específica o legacy (Oracle, SQL Server, etc), en la cual se utilicen procedimientos almacenados directamente en el motor, éstos deben ser securizados de forma correcta, por ejemplo, utilizando bind variables.

3. Codificar y escapar los datos

Éstas son técnicas defensivas, cuyo objetivo es detener ataques de inyección, ya sean SQL, XSS u otros. Codificar consiste en transformar o traducir ciertos caracteres especiales en otros caracteres equivalentes pero inofensivos para el intérprete. Por ejemplo, el carácter “<” se traduce en “<”.

Escapar caracteres es una técnica similar, con la diferencia que en lugar de reemplazar un carácter por otro, se añade un carácter especial antes del dato a escapar, para prevenir interpretaciones erróneas, por ejemplo, añadiendo un “\” antes de caracteres como [“] (comillas dobles), para que sea interpretado como texto y no como el cierre de un string.

Es fundamental realizar este escapado y/o codificación de datos en un entorno confiable (en el servidor y no en el navegador del usuario), de forma tal, de evitar el “doble escapado” que genera errores de despliegue de los datos (por ejemplo, si escapamos antes de almacenar en la base de datos y nuevamente cuando se despliega al usuario).

Para evitar ataques XSS se deben utilizar los mecanismos conocidos como Contextual Output Encoding, que nos permiten prevenir el despliegue de contenido malicioso en el navegador del usuario.

Existen otros tipos de codificación y escape, tales como shell escaping, XML escaping, LDAP escaping y otros, que podrán ser utilizados en la medida que sea necesario.

Un sistema debe considerar como potencialmente inseguros todos los datos que provengan desde fuera de la aplicación, así como también los datos que provienen desde la base de datos (en caso que se haya modificado

maliciosamente la misma). Por ello, deben ser correctamente codificados y/o escapados siempre, pero sin olvidar el contexto de los datos, por ejemplo, si es HTML, datos alfanuméricos u otros.

Se deben considerar también las validaciones de datos, descritas a continuación.

4. Validación de datos

Ésta es una técnica para asegurar que sólo los datos con el formato correcto podrán ingresar al sistema a desarrollar. Esta validación debe ser correcta, tanto sintáctica como semánticamente. Por ejemplo, si esperamos que en un campo se puedan ingresar sólo dígitos, no debemos aceptar otro tipo de caracteres (validación sintáctica). La validación semántica es un poco más compleja y consiste en validar que los datos tengan sentido en el contexto de la aplicación, por ejemplo, una fecha de inicio no podría estar después de una fecha de fin.

La validación de datos debe ser, como mínimo, a través de mecanismos de lista negra (datos conocidos como maliciosos). Como una mejor práctica también se validará a través de lista blanca (datos conocidos como correctos, por ejemplo, una lista de regiones).

La validación de datos, al igual que la codificación y escape, deben ser siempre realizados en el servidor y nunca del lado del cliente (por ejemplo, no se debe realizar en el navegador del usuario). No se debe confundir un aviso al usuario (por ejemplo, para alertar de un campo mal ingresado) con la validación de datos.

La validación de datos no convierte automáticamente a los datos en seguros, por lo que se debe utilizar en conjunto con otras defensas, tales como la parametrización de consultas y escapado de datos, mencionados anteriormente.

Para la validación de datos, incluyendo datos complejos, tales datos serializados, HTML u otros, se deberían utilizar librerías apropiadas para ello (HTML Purifier, Bleach, entre otras).

5. Autenticación nivel 1: Contraseñas

Las aplicaciones deben exigir al usuario el uso de contraseñas de características y calidad adecuadas, y que no sean contraseñas previamente filtradas[1]. Asimismo, se deben implementar mecanismos seguros de recuperación de contraseñas, ya sea utilizando un segundo factor de autenticación o a través de canales diferentes (como teléfono móvil o correo electrónico).

Una contraseña fuerte es aquélla que es difícil de detectar, tanto por humanos

[1] Por ejemplo, las de este link.

como por software, protegiendo efectivamente los datos de un acceso no autorizado.

Una contraseña segura consta de al menos ocho (8) caracteres (y mientras más caracteres, más fuerte es la contraseña), que son una combinación de letras, números, símbolos y el uso de mayúsculas y minúsculas. Ésta no debe contener palabras que se pueden encontrar en un diccionario o partes del nombre del usuario.

Para almacenar las contraseñas, se deben utilizar algoritmos criptográficos especialmente diseñados para este fin, tales como bcrypt, PBKDF2 y Argon2. Se debe evitar utilizar algoritmos de hash "a secas", tales como SHA-1, SHA-2 y MD5 para el almacenamiento de contraseñas.

También se deben implementar mecanismos de limitación de intentos de inicio de sesión, ya sea ralentizando los intentos de inicio de sesión, bloqueando la IP de origen de las pruebas fallidas o bloqueando las cuentas después de un número predeterminado de intentos fallidos. Esta estrategia no debe bloquear permanentemente las cuentas, puesto que esto podría causar una denegación de servicio a usuarios legítimos.

En el caso de que los ataques de fuerza bruta sean un problema, es recomendado implementar múltiples factores de autenticación.

6. Autenticación nivel 2: Multifactor

Es el uso de múltiples factores de autenticación para verificar al usuario. Habitualmente se usa una combinación de dos o más de los siguientes factores:

- Lo que el usuario sabe (por ejemplo, una contraseña)
- Lo que el usuario tiene (por ejemplo, un dispositivo o llaves de seguridad)
- Algo que el usuario es (por ejemplo, la biometría).

La biometría no se debe considerar un dato secreto, puesto que algunas características biométricas pueden ser fácilmente reproducibles u obtenidas sin conocimiento del usuario, por ejemplo, a través de una foto del rostro o "levantando" huellas digitales de objetos, por lo que siempre debe usarse en conjunto con otro factor de autenticación.

7. Autenticación nivel 3: Criptográfica

Se logra a través del uso de módulos de hardware criptográfico seguro, en conjunto con algún mecanismo de autenticación adicional (password, biometría, otros).

8. Implementar mecanismos de manejo de sesión

Para el manejo de las sesiones, se debe considerar al menos lo siguiente:

- El identificador de sesión debe ser único, suficientemente largo y aleatorio.
- Se deben generar (o rotar) los identificadores de sesión durante la autenticación y re-autenticación.
- Se debe implementar un timeout por inactividad que fuerce la re-autenticación al usuario. La duración de este timeout debe ser inversamente proporcional a la sensibilidad de los datos a proteger, vale decir, mientras más sensible, menor duración.

9. Uso de cookies para manejo de sesión

Para el uso de cookies se debe considerar lo siguiente:

- Deben ser accesibles por el mínimo de dominios requeridos para el correcto funcionamiento del sistema.
- Deben caducar al momento en que expira la sesión, o luego de un corto período.
- Deben tener el flag *"secure"*. Esto fuerza su transferencia a través de TLS.
- Deben tener el flag *"HttpOnly"*. Esto previene su acceso a través de JavaScript.

10. Uso de tokens de sesión

Cuando sea necesario manejar sesiones stateless, por razones de performance u otras, se recomienda el uso de JWT (JSON Web Tokens), puesto que es un mecanismo seguro e interoperable de manejo de sesión.

Un JWT[2] tiene garantías criptográficas si es generado de forma segura, para lo cual se recomienda utilizar los siguientes parámetros:

- Nunca definir, en la configuración, el algoritmo para firmar o cifrar como "none". Esto deshabilita todas las consideraciones criptográficas. El parámetro *alg* es el que permite definir el algoritmo que será utilizado para estas tareas.
- En relación al punto anterior, se debe seleccionar un algoritmo lo suficientemente fuerte.
- Se deben generar los secretos (o secrets) de forma criptográficamente segura, con una correcta entropía por parte del servidor que los genere.
- Rotar periódicamente, por ejemplo, cada 3 meses, los secretos

[2] https://tools.ietf.org/html/rfc7519

utilizados para firmar los tokens de sesión.

11. Identificación de datos

Los datos sensibles o críticos deben estar identificados para poder implementar las protecciones que requieran de forma correcta. Por ejemplo, si se manejan datos sensibles de ciudadanos y se requiere almacenarlos bajo encriptación.

12. Datos en tránsito

Las comunicaciones de los componentes que transporten información de los usuarios entre sistemas, deberán siempre estar protegidas mediante TLS, idealmente la versión 1.3, y los sistemas correctamente configurados para seleccionar el cifrado más fuerte disponible.

13. Datos en reposo

Se debe evitar, cuando sea posible, el almacenamiento de datos sensibles por parte de la aplicación.

En caso que no se pueda evitar almacenar datos sensibles, éstos deben ser protegidos mediante encriptación, para prevenir la modificación o acceso no autorizado.

Considerando que la encriptación es un tema altamente complejo, en particular las buenas prácticas, mecanismos de cifrado y gestión de llaves.

En caso de manejar datos que deban considerarse como abiertos, éstos deberán ser manejados y almacenados utilizando formatos estándar que permitan su fácil exportación a las plataformas correspondientes, por el portal datos.gob.cl.

14. Manejo de secretos

Las aplicaciones habitualmente contienen múltiples secretos que son necesarios para su operación. Éstos pueden incluir certificados digitales, contraseñas para la base de datos, credenciales a otros servicios y llaves criptográficas, entre otros. Toda la información relacionada a certificados digitales, contraseñas, llaves, credenciales, incluidas las rutas de almacenamiento u otras referencias a estos objetos, deben quedar debidamente parametrizadas en un archivo de variables de entorno y éste debe ser excluido de la herramienta de control de versiones.

15. Desarrollo orientado a pruebas

Para efectos de trabajo con datos o ejercicios, el equipo de desarrollo deberá tener un set de datos no válidos, escogido para trabajar y cargar el sitio para efectos de pruebas de integración continua o pruebas unitarias. Este set

no debe contener información que sea real y la cantidad de datos debe ser reducida, pero suficiente para generar pruebas sobre el código.

16. Calidad de código

El código deberá ser revisado de forma continua durante su construcción con herramientas de inspección.

Ejemplos de herramientas son:

- CodeClimate
- SonarQube
- Github Advanced Security

El objetivo principal es encontrar posibles bugs de seguridad, tanto dentro del código creado por el equipo de desarrollo, como en sus dependencias. La configuración de ambas, o al menos de una de las aplicaciones, debe ser guardada como archivo de configuración dentro del código, para luego ser usado en la integración continua.

17. Detección preventiva

Para todo lo relacionado con las aplicaciones, se debe configurar y usar algún proyecto OWASP (ejemplo: OWASP ZAP) para la búsqueda de vulnerabilidades en la aplicación. Esto, para corregir lo que sea necesario antes de entregar o desplegar la aplicación. Sin embargo, como se considera un desarrollo continuo después de una entrega final, la configuración del proyecto usado debe quedar documentado dentro del código, tanto para uso futuro como para ser integrada dentro del pipeline de despliegue/integración continua.

18. Modelo inicial de datos

Las estructuras de base de datos no deberán contener datos, tan sólo los esquemas y, de requerir la carga de datos externos producto de un proceso de inicialización, estos datos no deben ir en el código y serán procesados por una vía interna y privada, en el caso que se requiera.

19. Seeding de la base de datos

La creación de usuarios administradores iniciales o de cualquier tipo de información sensible, debe quedar fuera del sistema de versionamiento del código.

El proceso de creación de un usuario inicial de administración, así como cualquier otro dato sensible para el funcionamiento de la aplicación cuya filtración involucre una merma de seguridad, debe ser documentado, entregado confidencialmente y nunca ser registrado en un repositorio de código.

20. Implementar mecanismos de registros o *logs*

Se deben registrar distintos eventos del sistema para permitir que éstos sean monitoreados de forma automatizada para efectos de seguridad:

- Utilizar un formato común de registros al interior de la institución. Esto permite facilitar la configuración y parametrización de los mecanismos de monitoreo.
- Registrar la cantidad de información correcta.
- Siempre registrar un timestamp e información de identificación (IP, usuario, etc).
- Nunca registrar información sensible en los logs.

Dentro del stack de herramientas detallado en Tecnologías de Preferencia, se encuentran opciones Open Source.

21. Manejo seguro de errores y excepciones

Se debe desarrollar el sistema, de forma tal, que aplique el principio de "fallar seguro", es decir:

- No exponer información sensible o privada en los mensajes de error. En particular, nunca olvidar desactivar el modo debug al pasar a producción.
- Asegurarse de que una excepción o fallo no comprometa la seguridad por un error de programación en el sistema. Por ejemplo, causar una denegación de servicio o ejecución de código con privilegios incorrectos.
- Registrar las excepciones adecuadamente en los sistemas que correspondan.

22. Compilación limpia

De ser un lenguaje compilado y no interpretado, no es necesario contar con ningún tipo de alerta en el momento de compilación. Para conseguir este objetivo, lo mejor es dar las opciones al compilador para tratar las alertas (*warnings*) como si fueran errores y, de esta forma, fallar en caso de existir una alerta (*warning*) al momento de la compilación.

IV. Aseguramiento y certificación de calidad

Para facilitar la ejecución y registro de pruebas se recomienda el uso de herramientas de integración/despliegue continuo (CI/CD) y de herramientas de revisión del código integradas a este pipeline CI/CD.

Se recomienda que los equipos de desarrollo de las instituciones cumplan con una estrategia de aseguramiento de la calidad, la cual como mínimo debe incluir:

- Plan de pruebas.
- Diseño de las pruebas.
- Registro de su ejecución y resultados.

El resultado de la ejecución de las pruebas sirve como control de calidad del software desplegado.

Para facilitar la ejecución y registro de pruebas se recomienda el uso de herramientas de integración/despliegue continuo (CI/CD) y de herramientas de revisión del código integradas a este pipeline CI/CD.

Al final de este documento se presentan algunos ejemplos de archivos de configuración de un pipeline CI/CD en las herramientas Gitlab y Github.

V. Tecnologías de preferencia

Para facilitar la ejecución y registro de pruebas se recomienda el uso de herramientas de integración/despliegue continuo (CI/CD) y de herramientas de revisión del código integradas a este pipeline CI/CD.

Se recomienda el uso de las siguientes tecnologías de desarrollo y arquitectura del software.

1. Lenguajes de programación

- Sistemas de información:
  - ➢ Python 3.6+
  - ➢ PHP 7.1
  - ➢ Java 11+
  - ➢ C# 7.3+
  - ➢ Ruby 2.4.9+
  - ➢ Go Language 1.13+
- Microservicios:
  - ➢ Python 3.6+
  - ➢ NodeJS 8+
  - ➢ Go Language 1.13+
  - ➢ Java 11+
  - ➢ C# 7.3+
- Los *frameworks* y bibliotecas para desarrollo recomendados son:
  - ➢ PHP: Laravel, Symfony, CodeIgniter
  - ➢ Python: Django y Flask
  - ➢ Ruby: Ruby on Rails
  - ➢ Go: Revel, Gin, Martini
  - ➢ Java: Spring Boot, Splunk
  - ➢ C#: .NET Core Framework
  - ➢ ReactNative
  - ➢ ExpressJS
- Bibliotecas gráficas y herramientas:
  - ➢ Bootstrap
  - ➢ D3JS
  - ➢ Sass, Less
  - ➢ Grunt
  - ➢ NPM
- Frameworks Javascript

- ➢ VueJS
- ➢ AngularJS
- ➢ ReactJS
- ➢ JQuery

2. Almacenamientos de datos relacionales

- PostgreSQL 10 o superior.
- MySQL 5.7 o superior.
- MSSQL 2016 o superior.
- Oracle 12c Release 2 o superior.

3. Almacenamientos de datos no relacionales

- MongoDB 4 o superior
- Elasticsearch 5.5 o superior
- Redis 5.0 o superior

4. Tareas de encolamiento

- RabbitMQ
- ZeroMQ

5. Tareas de registro eventos (*logging*)

- Elasticsearch 5.5 o superior
- Logstash
- Kibana
- Sentry

6. Despliegue de aplicaciones

- Kubernetes
- Helm
- Docker
- CloudFoundry

7. Para infraestructura como código

- Terraform

VI. Uso de tecnologías

Para facilitar la ejecución y registro de pruebas se recomienda el uso de herramientas de integración/despliegue continuo (CI/CD) y de herramientas de revisión del código integradas a este pipeline CI/CD.

1. Lenguaje

Al escoger el lenguaje de programación, se recomienda hacer uso de un framework para desarrollar bajo el paradigma de la programación orientada a objetos y utilizar un patrón de arquitectura tales como Modelo-Vista-Controlador (MVC), Microservicios u orientada al dominio (Hexagonal, Onion o Clean), según las necesidades del negocio y los atributos de calidad del software.

Algunos ejemplos de frameworks que soportan este paradigma en diversos lenguajes son: Laravel, Symfony, CodeIgniter, Flask, Django, Beego, Revel, Spring Boot, etc.

2. Almacenamiento

Para la gestión de datos, tanto relacionales como no relacionales, se sugiere instalar las bases de datos en modo cluster. Además, se deben tomar las precauciones necesarias, por ejemplo, haber creado los índices apropiados, al momento de crear las consultas y hacer uso de las mismas.

En el caso de las bases de datos relacionales, se debe considerar, entre otros:

- Tiempo de respuesta; puede variar si un cluster es muy grande.
- No tener referencias circulares.

Con respecto al uso de bases de datos no relacionales, se debe tener cuidado en la elección del driver, debiendo ser capaz de conectarse a más de un nodo del cluster a la vez para, de esta forma, mantener el paradigma de un sistema en cluster y capaz de crecer de forma horizontal.

3. Tareas fuera de línea o programadas

Para las tareas asíncronas o que tengan la característica de una tarea programada diaria o de tiempo definido, éstas deben usar tecnologías de encolamiento, como las que se mencionan en Tecnologías de

Preferencia. El consumo de las colas debe ser con procesos que no tengan relación con el sitio y debe ser posible desarrollarlos e implementarlos como un servicio separado.

4. Formateo de código

Para una mejor lectura del código por personas en la organización y para el futuro, se recomienda de sobremanera el uso de los formateadores de texto.

Éstas son herramientas que permiten limitar el largo de las líneas, lugares de llaves y paréntesis, así como también ayudan a ordenar el código , lo que debe ser una tarea constante.

Estas herramientas pueden ser configuradas en el editor de texto preferido de los programadores. Para lenguajes como Python, se sugiere PEP8, sin embargo, el estilo y formateo de una organización debe ser elegido por la misma y se sugieren los estándares propuestos para cada lenguaje por sus creadores.

5. Repositorios de código

Se debe utilizar repositorios para almacenar el código de las aplicaciones, ya sea on premise o mediante algún servicio en la nube. Github, Gitlab y Bitbucket son algunos ejemplos de servicios de repositorio de código.

VII. Desarrollo y consumo de APIs

La exposición de los servicios API REST debe utilizar mecanismos de autenticación para su consumo privado, ya sea tokens (JWT u otros), certificados o llaves criptográficas, u otros como medios de autenticación entre puntos.

1. Ley de Transformación Digital (21.180)

Actualmente, en el marco de la Ley de Transformación Digital (21.180) se está elaborando una Norma Técnica de Interoperabilidad. Esta Norma Técnica consignará los estándares de interoperabilidad que deberán cumplir las instituciones.

Mientras esta Norma Técnica se elabora y publica, a continuación, se mencionan algunas buenas prácticas para aquellas instituciones que están utilizando arquitectura REST en sus servicios web.

2. Desacoplar clientes y servicios web

Las aplicaciones construidas bajo los lineamientos expuestos hasta acá, deberán propender a cumplir con dos características fundamentales:

- Independencia de la plataforma.
- Evolución del servicio.

Para cumplir esto, se deben desacoplar las implementaciones de clientes y servicios, y poner a disposición sus métodos y operaciones.

3. Uso de HTTP

Las operaciones de las interfaces programables (APIs) deben tender a utilizar el protocolo HTTP[3] , ser procesadas en tiempo real y entregar una respuesta según la codificación establecida en el protocolo HTTP:

- GET: Solicitar un recurso.
- POST: Crear un nuevo recurso subordinado dentro de una URL existente.
- DELETE: Eliminar un recurso.
- PUT: Crear un nuevo recurso a una nueva URL o modificar un recurso existente en una URL.

[3] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

- HEAD: Idéntica a GET, excepto que no se retorna un cuerpo del mensaje en la respuesta.
- CONNECT: Establece un túnel hacia el servidor identificado por el recurso.
- OPTIONS: Utilizado para describir las opciones de comunicación para el recurso de destino.
- TRACE: Realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.
- PATCH: Es utilizado para aplicar modificaciones parciales a un recurso.

Es importante identificar que, de los métodos precedentes, los cuatro primeros son los más utilizados, en particular para atender las operaciones CRUD (create, read, update y delete).

4. Exposición y autenticación de servicios

La exposición de los servicios API REST debe utilizar mecanismos de autenticación para su consumo privado, ya sea tokens (JWT u otros), certificados o llaves criptográficas, u otros como medios de autenticación entre puntos.

5. Definición de URI de recursos

La definición de URI de recursos debe propender a seguir las prácticas expresadas a continuación como ejemplo:

- Usar sustantivos para describir los recursos:
  - ➢ GET /usuarios/ lista todos los usuarios
  - ➢ GET /usuarios/12 muestra detalle del usuario con id 12
  - ➢ POST /usuarios crear usuario
  - ➢ PUT /usuarios/12 actualiza usuario con id 12
  - ➢ PATCH /usuarios/12 actualiza parcialmente usuario con id 12
  - ➢ DELETE /usuarios/12 borra usuario con id 12
- Usar "/" para indicar la relación de jerarquía:
  - ➢ GET/ usuarios/12/mensajes muestra mensajes del usuario con id 12
  - ➢ GET/usuarios/12/mensajes/93 obtiene mensaje con id 93 del usuario con id 12
  - ➢ POST /usuarios/12/mensajes crear mensaje para usuario con id 12
  - ➢ DELETE /usuarios/12/mensajes/3 elimina mensaje 3 de usuario con id 12
- Usar parámetros de consulta (query strings) para realizar operaciones de filtro, ordenamiento, paginación y/o búsqueda:

- ➢ GET /instituciones/buscar?nombre=ministerio* busca organizaciones cuyo nombre comienza con “ministerio”
- ➢ GET /instituciones/2/usuarios?estado=inactivo lista todos los usuarios de la institución 2 con estado inactivo

- Formato de las URIs:
  - ➢ No usar “/” al final de la URI.
  - ➢ No usar mayúsculas.

    /instituciones/2/usuarios

    /Instituciones/2/Usuarios/

- Usar guión “-” (y no guión bajo “_”), más conocido como kebab-case, para facilitar la comprensión de la URI en los casos en que sea más de una palabra para el servicio.

  /oficinas/134/como-llegar

  /oficinas/134/como_llegar

- Usar sólo letras en minúscula dentro de la URI. No utilizar caracteres especiales ni acentos.
- Versionamiento de los servicios. Se sugiere contar con un número de versión en la URI, el cual deberá ser incrementado en la medida en que el servicio se vea modificado en su naturaleza o los datos que entrega.

  URI Versión 1 antigua:

  GET /v1/oficinas/134/como-llegar

  URI Version 2 actual:

  GET /v2/oficinas/134/como-llegar

- No usar extensiones de archivos.

  /instituciones/2/descripcion

  /instituciones/2/descripcion.txt

  /instituciones/2/descripcion.json

Para la descripción de servicios se recomienda utilizar el estándar OpenAPI Specification (OAS) v3.0+, siendo este punto un requisito fundamental en la documentación entregable de un proyecto.

6. Uso del idioma en el código

Se recomienda utilizar de la siguiente forma el uso de idioma castellano e inglés a nivel de programación y definición de esquemas para la construcción de servicios:

Castellano

- Variables y contenido
- Construcción de las URIs
- Documentación
- Metadatos
- Tablas de bases de datos

Inglés

- Operaciones, por ejemplo, métodos get, create, delete, etc.

- Campos de auditorías en base de datos, por ejemplo, created_at, updated_at y deleted_at, etc.
- Encabezados

VIII. Uso de contenedores en el despliegue

Para el despliegue de las aplicaciones es altamente recomendado el uso de contenedores, debido a la facilidad de movilidad de los mismos y replicación de los distintos ambientes. Dada su característica de idempotente, nos permiten replicar en una plataforma para crecer de forma horizontal.

Para el despliegue de las aplicaciones es altamente recomendado el uso de contenedores, debido a la facilidad de movilidad de los mismos y replicación de los distintos ambientes. Dada su característica de idempotente, nos permiten replicar en una plataforma para crecer de forma horizontal.

En plataformas con contenedores Docker, el uso de herramientas para la gestión y automatización es muy recomendable. En este caso, se sugiere usar Ansible, Chef o Puppet para la administración de los servidores y levantar los contenedores en los mismos, utilizando la herramienta docker-compose provista por Docker. Este set de herramientas permite su fácil y rápida replicación.

Para situaciones en las cuales se puede contar con cluster Kubernetes o CloudFoundry, las herramientas provistas por cada uno es suficiente para el despliegue, ya que estas herramientas están preparadas para la gestión, replicación y respaldo de los ambientes, y es posible generar más de un ambiente usando las herramientas que ofrecen para ello

Se entiende también que muchas veces las aplicaciones no pueden ingresar a un contenedor, o por su carácter legacy no es posible introducirlas en estos contenedores idempotentes. Sin embargo, se sugiere considerar su uso en los futuros desarrollos y, de esta forma, ir avanzando hacia infraestructuras más flexibles y que pueden crecer de forma horizontal.

IX. Licencias

Todo desarrollo realizado al interior del Estado debe propender a estar licenciado, acorde a las necesidades de uso con que fue creado.

Todo desarrollo realizado al interior del Estado debe propender a estar licenciado, acorde a las necesidades de uso con que fue creado. Para el Estado es fundamental la colaboración entre instituciones, por lo que se sugiere la construcción de software cuyo código fuente sea accesible por otras instituciones, así como también para estar frente al escrutinio de los ciudadanos y que éstos puedan realizar los aportes que consideren necesarios para mejorar el código de aplicaciones que ellos mismos pueden usar.

Para estos efectos, están a disposición las siguientes licencias:

1. GPLv2 y GPLv3

Este conjunto de licencias es conocido como copyleft, es decir, requiere que las modificaciones realizadas al software, incluyendo cambios efectuados por terceros, sean puestos a disposición de los receptores del software utilizando la misma licencia. Esto impulsa la colaboración entre instituciones y da la posibilidad de que todas ellas puedan aportar a las mejoras del software y, aquellas que no tengan la capacidad de desarrollo, puedan usarlo sin problemas futuros.

Para todos los desarrollos, se recomienda el uso de la licencia GPLv3, pero también se puede utilizar la licencia GPLv2 en caso de que alguna institución requiera modificar software que utilice dicha licencia y no pueda apelar a re-licenciar la misma a partir de una fecha específica.

2. LGPLv3

Esta licencia sirve cuando el software debe ser enlazado con módulos que son privativos o no compatibles con alguna de las versiones de GPL.

3. Apache 2.0

Esta licencia, compatible con GPLv3 (no inferiores), supone muchas ventajas para mantener el código libre y también permite trabajos secundarios y uso del código en otros proyectos. Se recomienda su uso en todo proyecto que requiera de una cierta interacción

con el mundo privado y cuando esta interacción sea a largo tiempo. Un ejemplo de esto último es cuando se genera un producto complementario que podría llegar a ser usado por otros gobiernos e, incluso, por el mundo privado.

4. Dominio Público (Creative Commons 0 y equivalentes)

Estas licencias actúan como el equivalente a poner en dominio público el trabajo realizado. Son eficaces para compartir datos que puedan ser utilizados por la comunidad científica, medios audiovisuales y otros datos que requieran una distribución lo más amplia posible.

Habitualmente, esta licencia se aplica a los datos generados por un software y no al software en sí, salvo algunas excepciones.

5. Casos de uso de licencia

Al momento de aplicar las licencias se debe tener especial consideración en cómo se desea perpetuar el código en el tiempo, es decir, cuando el código sea un proyecto relevante, éste debe considerar que al estar público da lugar a que la comunidad tome el código y pueda hacer cambios en el mismo, para uso propio o para vender de vuelta estos cambios al Estado.

Sin embargo, al usar una licencia GPLv3 esto se puede evitar, ya que fuerza a que los cambios mayores realizados al código sean contribuidos al proyecto principal y, de esta forma, mantener todo cambio en el código libre en el tiempo.

Al dejar el código en dominio público, sin licencia o usando una licencia clásica Creative Commons, no se está buscando que los cambios al código vuelvan a la base sino, más bien, mantener la autoría del mismo. De esta forma, toda persona dentro de la comunidad puede tomar el código y realizarle cambios, cerrarlo y utilizarlo para prestar servicios al Estado, usando el código del Estado. Es por ello que estas licencias no son recomendadas para software de carácter crítico o que su uso pueda masificarse de forma importante en el tiempo.

Por otro lado, es necesario considerar que muchas veces el software podría estar usando módulos con licencias no compatibles con una GPLv3 o GPLv2. Por esto, el uso de LGPLv3 es recomendado, ya que permite incorporar módulos propietarios. Este caso se da para aplicaciones que son compiladas y enlazadas con otros módulos. No aplica para lenguajes interpretados.

6. Buenas prácticas de licenciamiento

Al momento de licenciar el software es necesario agregar el texto exacto de la licencia usada en un archivo llamado LICENSE, COPYING, LICENSE.TXT o

COPYING.TXT. Este texto no debe ser modificado ni alterado de forma alguna, ya que vivirá con el código y será transportado por todo medio al momento de desplegar el mismo.

Es deseable, pero no mandatorio, incluir al principio de cada archivo del código un aviso de licenciamiento (copyright notice) con un extracto sugerido en la licencia. En el caso de la licencia GPLv3, debe ser de la siguiente forma:

N*ombre de aplicación*

Copyright (C) 2021 División de Gobierno Digital

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

X. Actualizaciones a esta guía

Dado que el mundo de la tecnología avanza cada vez a pasos más y más agigantados, esta guía será sometida a revisión una vez al año, en busca de las actualizaciones que puedan ocurrir, de forma de mantenerse actualizada.

Dado que el mundo de la tecnología avanza cada vez a pasos más y más agigantados, esta guía será sometida a revisión una vez al año, en busca de las actualizaciones que puedan ocurrir, de forma de mantenerse actualizada. Esta revisión será realizada por la División de Gobierno Digital en colaboración con todas las instituciones que deseen participar. Con esto se espera incluir todas las mejoras y aprendizajes de las instituciones conforme han seguido esta guía y pudieran encontrar mejoras a la misma.

Con respecto a la revisión por parte de la sociedad civil, se generarán instancias en las cuales la sociedad civil podrá realizar una revisión y evaluación para integrar nuevas mejoras a la guía, según su propia opinión y experiencia en el mundo de las tecnologías.

Sin desmedro de lo anterior, la guía a futuro podrá contener un anexo con fe de erratas apropiadas, que serán incorporadas en la medida que éstas sean detectadas, sin la necesidad de esperar a la revisión anual pública antes mencionada.

XI. Ejemplos

1. Ejemplo Dockerfile aplicación PHP

El archivo Dockerfile es el que da el inicio a la construcción de la imagen que será cargada en un servicio de registry de imágenes para su posterior uso. A continuación se muestra un Dockerfile de ejemplo de una aplicación PHP.

2. Ejemplo Gitlab-ci.yml

El archivo gitlab-ci.yml es un archivo YAML que se encuentra en la raíz del proyecto, y que se ejecuta automáticamente cada vez que se ejecuta un commit. Este archivo hace que un runner procese las tareas especificadas en el mismo archivo.

3. Ejemplo Github-action.yml

El mismo concepto que el gitlab-ci.yml, pero para Github actions

Ejemplo Dockerfile aplicación PHP

```
FROM php:7.1-fpm
ARG CREDENTIALS_GIT
ARG REPO=git.gob.cl/chileatiende/chileatiende
ARG DIRECTORY_PROJECT=/var/www/chileatiende

WORKDIR $DIRECTORY_PROJECT

#23 Install Packages
RUN apt-get update && apt-get install -y \
     libxml2-dev \
     git \
     zip \
     unzip \
     zlib1g-dev \
     libpng-dev \
     --no-install-recommends \
  #23 Docker extension install
  && docker-php-ext-install \
          opcache \
          pdo_mysql \
          pdo \
          mbstring \
          tokenizer \
          xml \
          ctype \
          json \
          zip \
          gd \
          bcmath \
  #23 error to stderr php-fpm
  && { \
      echo "log_errors = On" ; \
      echo "error_log = /dev/stderr" ; \
      echo "error_reporting = E_ALL" ; \
      echo "memory_limit = 256M" ; \
  } > /usr/local/etc/php/php.ini \
  #23 Install composer
  && curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer

#23 Clone repository
COPY . $DIRECTORY_PROJECT
  #23 Install dependencies from project
RUN composer install && composer clearcache && rm -rf /root/.composer /usr/local/bin/composer

#23 Permissions
RUN find $DIRECTORY_PROJECT -type f -exec chmod 644 {} \; \
  && find $DIRECTORY_PROJECT -type d -exec chmod 755 {} \; \
  && chown -R www-data:www-data $DIRECTORY_PROJECT \
  #23 Reduce image size
  && apt-get remove --purge -y git curl \
  && apt-get autoremove -y \
  && apt-get clean \
  && apt-get autoclean \
  && apt-get autoremove -y \
  && rm -rf /usr/share/locale/* \
  && rm -rf /var/cache/debconf/*-old \
  && rm -rf /var/lib/apt/lists/* \
  && rm -rf /usr/share/doc/* \
  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* /var/cache/apt/archives/*.deb /var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin

ENV LANG es_CL.UTF-8
ENV LANGUAGE es_CL:es
ENV LC_ALL es_CL.UTF-8
ENV TZ America/Santiago

RUN echo "APP_KEY=$(php artisan key:generate --show)"    > .env

EXPOSE 9000
CMD ["php-fpm"]
```

Ejemplo gitlab-ci.yml utilizando el registry de gitlab y con stage de code_quality

```
xxxxxxxxxx
image: docker:stable

stages:
    -   build
    -   code_quality

build:
stage: build
variables:
  DOCKER_DRIVER: overlay2
services:
    -   docker:stable-dind
script:
    -   docker login git.gob.cl:4567 -u gitlab-ci-token -p $CI_BUILD_TOKEN
    -   docker build -t git.gob.cl:4567/chileatiende/chileatiende:$CI_COMMIT_REF_NAME .
    -   docker push git.gob.cl:4567/chileatiende/chileatiende:$CI_COMMIT_REF_NAME
only:
    -   master
    -   staging

code_quality:
stage: code_quality
variables:
  DOCKER_DRIVER: overlay2
allow_failure: true
services:
    -   docker:stable-dind
script:
    -   export          SP_VERSION=$(echo          "$CI_SERVER_VERSION"          |          sed
        's/^\([0-9]*\)\.\([0-9]*\).*/\1-\2- stable/')
    -   docker run
          --env SOURCE_CODE="$PWD"
          --volume "$PWD":/code
          --volume /var/run/docker.sock:/var/run/docker.sock
            "registry.gitlab.com/gitlab-org/security-products/codequality:$SP_VERSION"  /code
artifacts:
  paths: [gl-code-quality-report.json]

build site:
stage: build
image: node:10-stretch
script:
    -   apt-get update
    -   apt-get -y install libpng16-16 libpng-tools libpng-dev
    -   npm i npm@latest -g
    -   npm install
    -   npm audit fix --force
    -   npm run prod
artifacts:
  expire_in: 30 days
  paths:
    -   public/*
```

Ejemplo github-actions.yml de deploy en un cluster AWS EKS utilizando el registry de AWS (ECR)

```
-   name: app_name
-   env:
-     AWS_ACCOUNT_ID:
-     REPOSITORY:
-     NAMESPACE:
-     ENVIRONMENT:
-     APP_NAME:
-     CLUSTER_NAME:
-   on:
-     workflow_dispatch:
-     push:
-       branches:
-         - "main"
-         - "develop"
-       tags:
-         - 'prod-*'
-         - 'develop-*'
-
-   jobs:
-     deploy:
-       runs-on: RUNNER_NAME
-       steps:
-       - uses: actions/checkout@v2
-
-       - name: set env
-         run: echo "TAG=${GITHUB_REF#refs/*/}" >> $GITHUB_ENV
-
-       - name: commit-sha para tag de imagen
-         run: echo "SHORT_SHA=`echo ${GITHUB_SHA} | cut -c 1-8`" >> $GITHUB_ENV
-
-       - name: build
-         run: |
              -   docker                                build                                -t
                  ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com/${REPOSITORY}:latest
                  -t
                  ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com/${REPOSITORY}:${SHOR
                  T_SHA} .
-       - name: upload image
-         run: |
              -   aws   ecr   get-login-password   --region   us-east-2   |   docker   login
                  --username        AWS_USERNAME                          --password-stdin
                  ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com
              -   docker                                                                    push
                  ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com/${REPOSITORY}:latest
              -   docker                                                                    push
                  ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com/${REPOSITORY}:${SHOR
                  T_SHA}
-       - name: kubernetes deploy
-         run: |
              -   aws eks update-kubeconfig --name ${CLUSTER_NAME} --region us-east-2
-                                        kubectl     set     image     deploy/$APP_NAME
    $APP_NAME=${AWS_ACCOUNT_ID}.dkr.ecr.us-east-2.amazonaws.com/${REPOSITORY}:${SHORT_SHA
    } -n ${NAMESPACE} --record
```