# Software architecture with Csharp 8 and .Net core

## Les tests :

**Les types de tests :**

* **Unit test :** *vérifie le comportement* d’une méthode ou *d’une* classe *isolée*.
* **Intégration tests :** *vérifie l’interaction de* plusieurs classes entre-elles.
* **Les tests bout en bout (end to end) :**  *vérifie le comportement de* l’application en testant un vrai scénario d’un utilisateur. Cela *permet de valider le fonctionnement du front et l’intégration du back*.
* **Tests d’acceptances :** *vérifie que les* critères d’acceptances / requirements *sont bien remplis*.
* **Test automatisés :** *test dont l’exécution ne* nécessite pas l’intervention d’un humain, de ce que je comprends *Selenium* est un framework pour développer ce type de test en *reproduisant un parcours d’utilisateurs en émulant son navigateur*.
  + **Exemple :** je vais sur cette page, je remplis x champs avec y valeurs, j’envois mon formulaire, je m’attends à avoir ça et ça.
* **Test de charge :**   *vérifie si l’application* peut gérer une charge d’utilisateurs *spécifiée* *pour un* certain *scénario.* L’application est *exécutée dans des* *conditions normales*.
* **Stress test / tests de stress :** *vérifie la* stabilité et le comportement *de l’application dans des* conditions extrêmes.
* **Test de contrainte :** *détermine* si *l’application avec un* module endéfaillance *peut renvoyer un résultat attendu*.

**Conseil :**

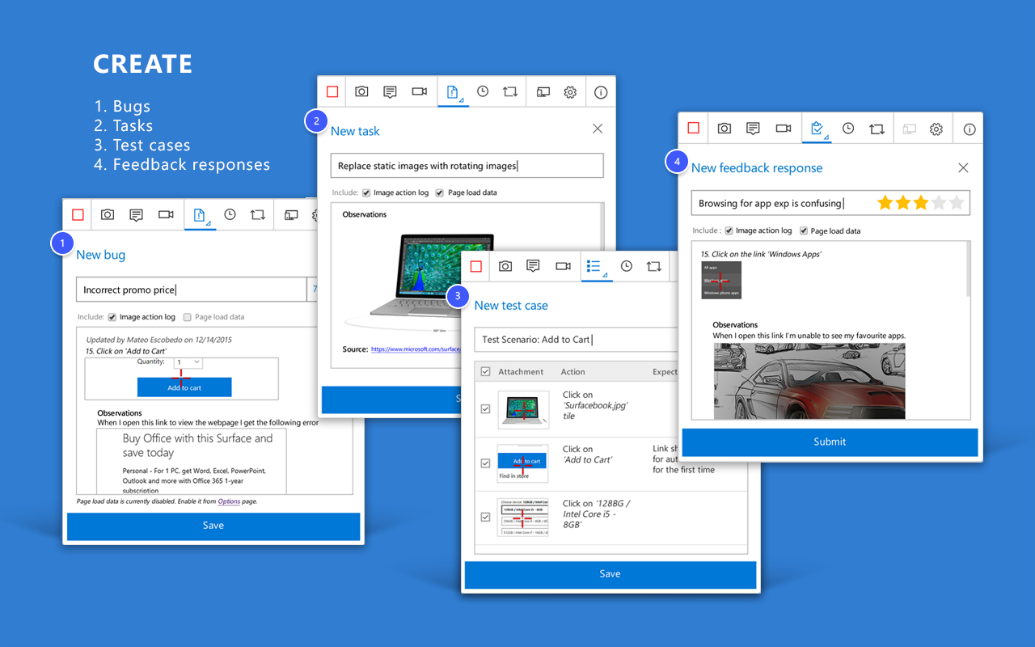
* Automatiser vos tests *pour* *vérifier l’intégrité de votre application après chaque modification*.
* *Utiliser des outils de reporting* pour avoir la *liste des tests qui sont passés, échoués, leur temps d’éxécution et leurs retours.*
* *Réaliser des tests de charges et de performances* en utilisant de fausses données de production *pour découvrir* *les* potentiels *bottleneck*, lorsque vous trouvez des lenteurs importantes, il faut *utiliser des outils qui nous renvoit des metrics pour trouver la cause du* *problème*.

**A savoir :**

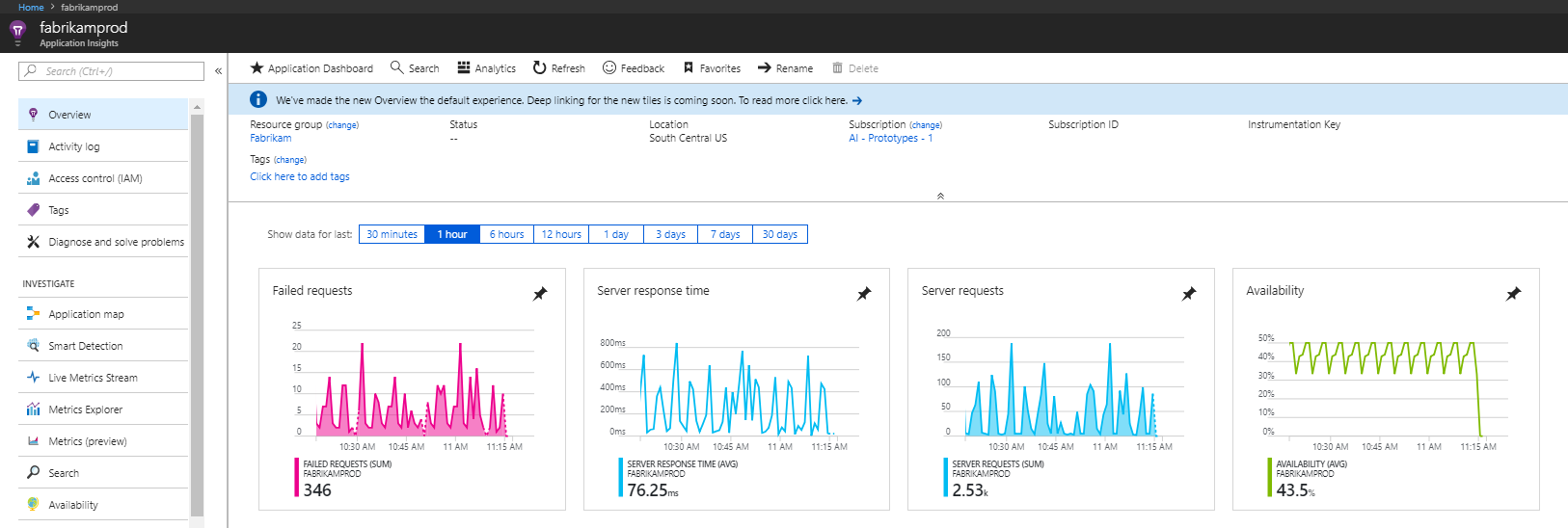
* *Utiliser des mocks créé de fausses implémentations de vos classes et ne permet donc pas de tester parfaitement le comportement de vos classes* et de vos méthodes ce qui fait qu’on peut passer à côté de pleins d’erreurs.

## Monitorer vos applications :

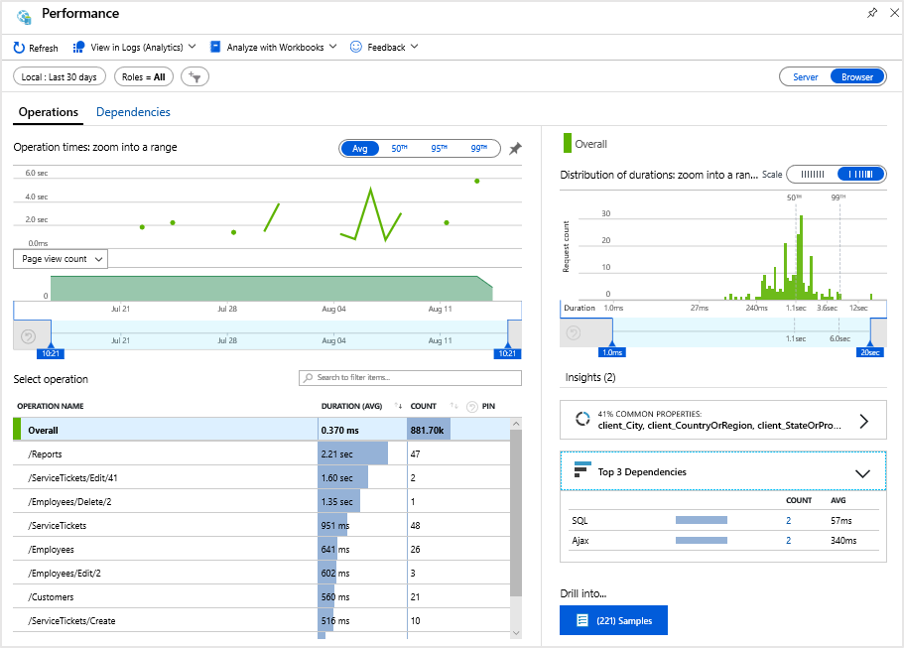
* **Healchecks**: *vérifier* *que* les différents composants de *votre* infrastructure *fonctionnent convenablement.*
* **SonarLint & Sonar Cloud :** *détecter les vulnérabilités et le* code *de faible qualité*.
* **Feedbacks de vos utilisateurs :** *leur envoyer des* formulaires pour récupérer leur *niveau* de *satisfaction* et leur dédié une *page* *de* reporting.
* **Feedbacks de votre équipe :** utiliser *l’extension* Test & Feedback qui permet de *créer* des *tâches*, des *bugs*, des cas de tests et *prendre et partager des vidéos de votre parcours utilisateur*, des *images* *et* du *texte*.

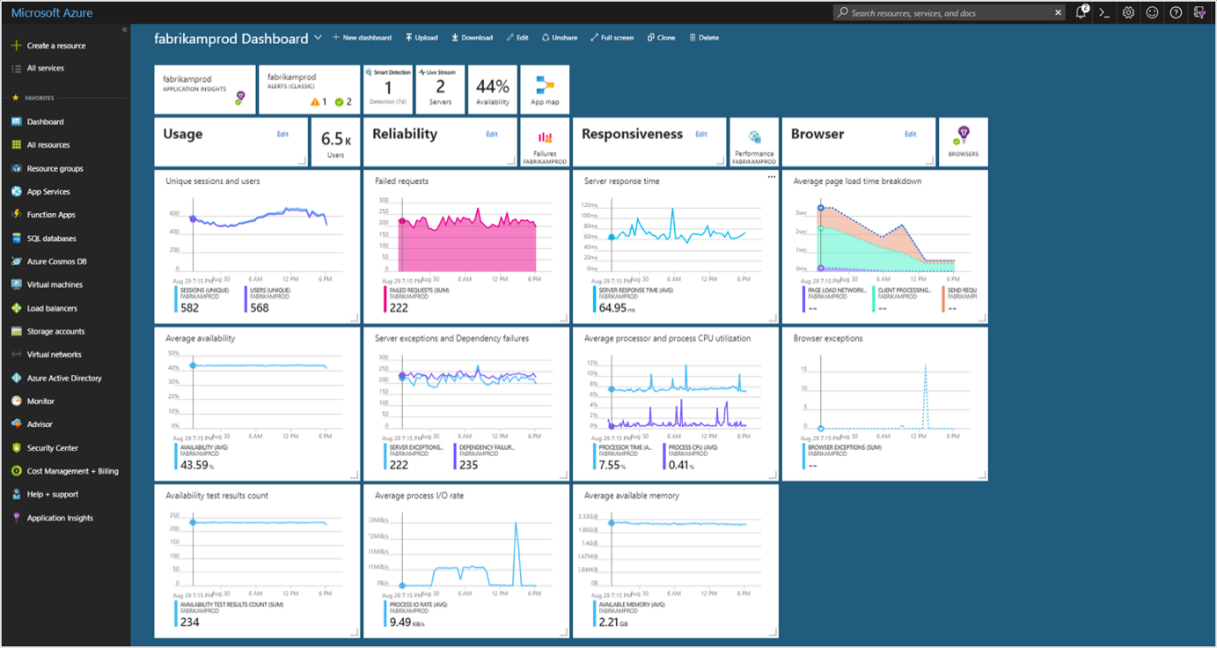


* **Des statistiques sur l’utilisation de votre application :** Insights.
  + **Créer un service de monitoring :** Azure > sélectionner votre *web* *app* > *insights* > enable.



* + Temps d’un utilisateur sur votre application, par où ils passent, le nombre de fois qu’un endpoint est appelé et son temps moyen calculé par unité de temps (jour, semaine, mois). Monitorer toutes les requêtes, les erreurs, le nombre d’appel par endpoint et la durée moyenne.





## Les outils d’analyse syntaxique / Code quality :

### Provenant de Visual Studio :

**Visual Studio > Analyse > Calculate code metrics :** afficher des *metrics* relative au *clean code*.

* **Depth of inheritance**: *hériter de classe créée des couplages*, au lieu d’utiliser de l’héritage *il faut utiliser* la composition (*l’aggrégation*) *et l’injection de dépendances*.
* **Maintenability index [1-100]:** prend en compte le *respect de la responsabilité unique*, le *code dupliqué*, les *classes et méthodes trop grosses*.
* **Cyclomatic complexity :** le *nombre de chemin* disponibles *que votre code peut prendre* (if / else / loop inside loop).
* **Lines of code :** *un nombre de ligne important* dans une classe ou une méthode *indique une mauvaise conception* dans le code.

**Visual Studio > Tools > Code Style :** *définir* votre *code style*.

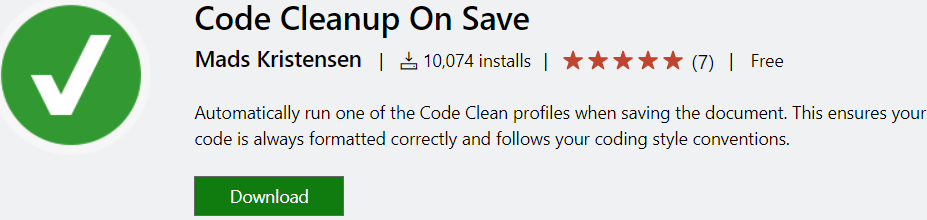
* **Considérer comme erreur / suggestion / warning des parties de notre standard :**

Une image contenant capture d’écran

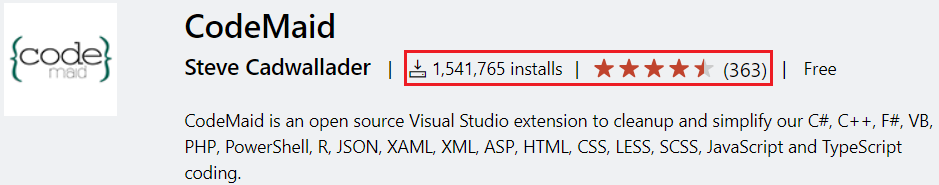
Description générée automatiquement

**Nettoyer votre code :**

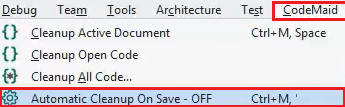
* **Visual Studio > Analyze > Code cleanup :** laisser un outil cleaner votre code.
  + **Nettoyer votre code automatiquement lors d’une sauvegarde :**



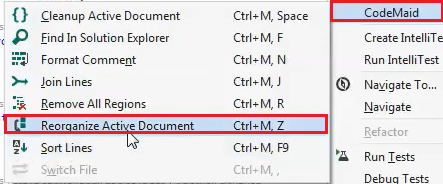
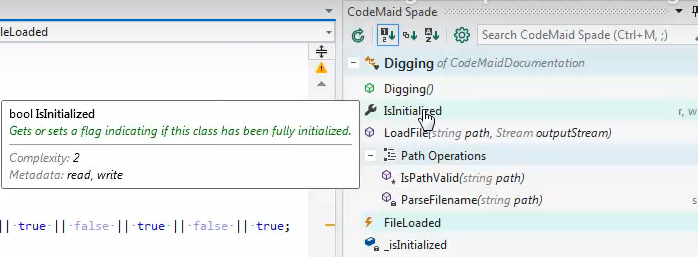
### L’extension Code Maid :



**Nettoyer votre code automatiquement lors d’une sauvegarde :**Code Maid > Automatically clean up on save.

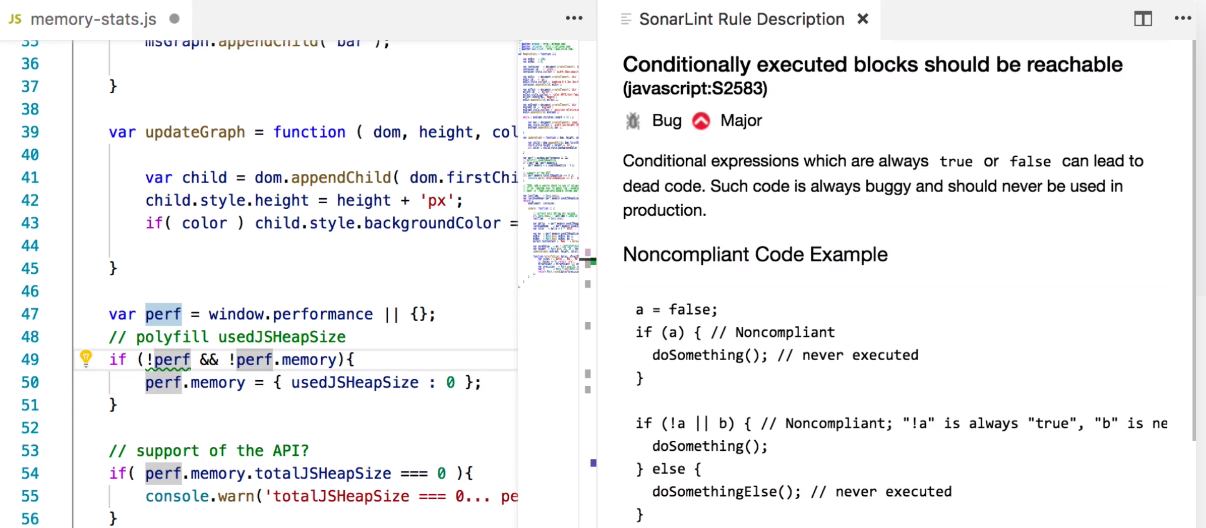


**Organiser et naviguer plus facilement entre les propriétés et les méthodes de votre objet :**



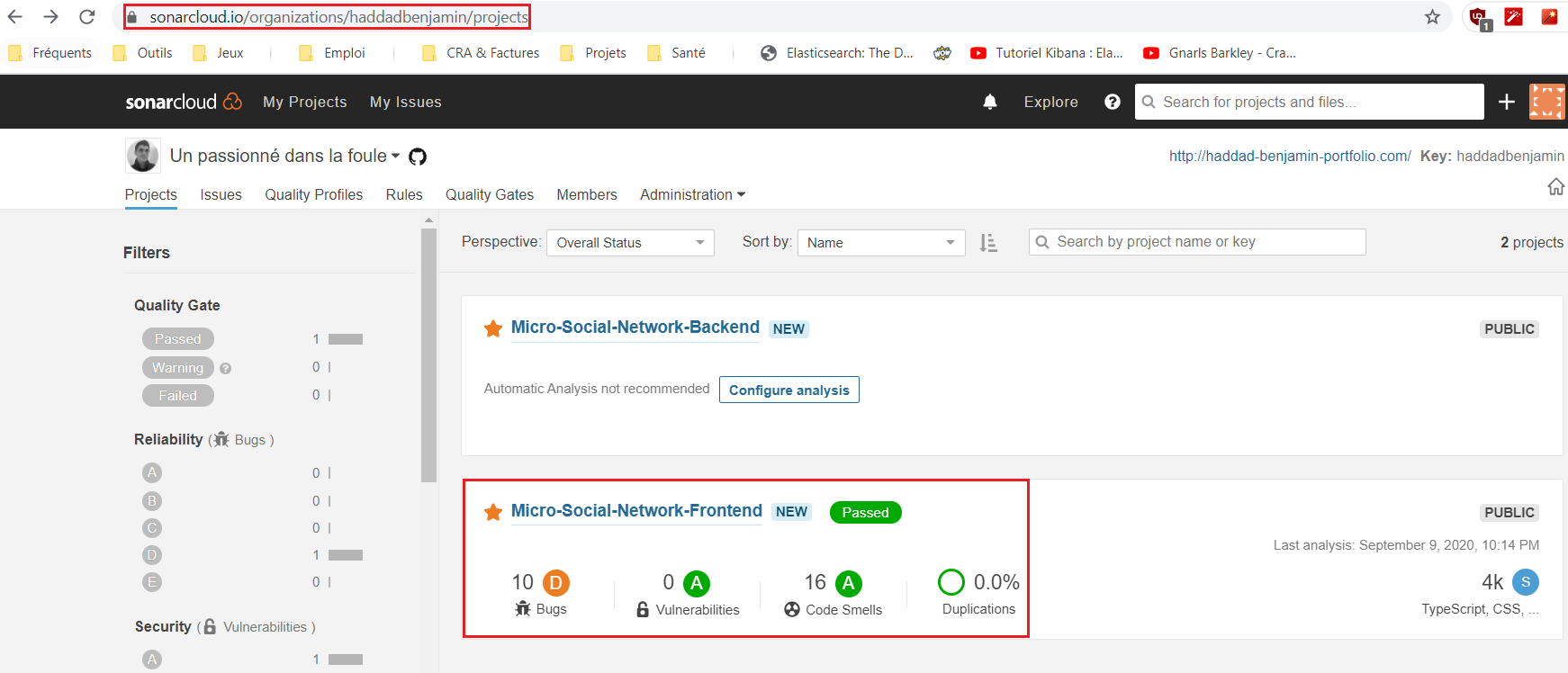
### L’extension SonarLint :

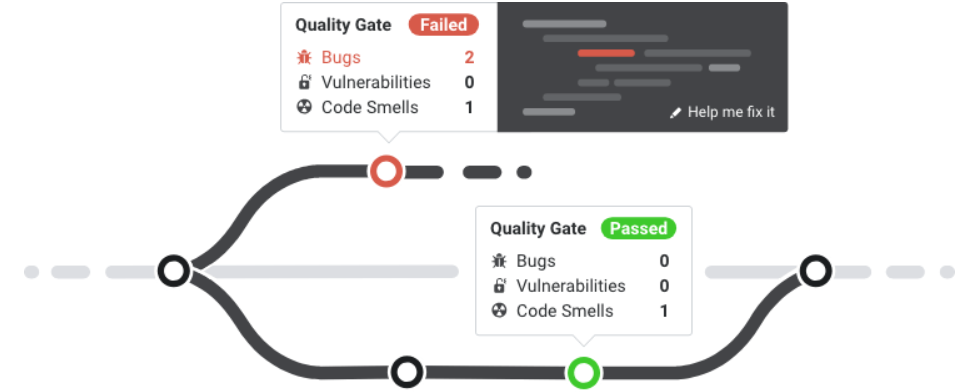
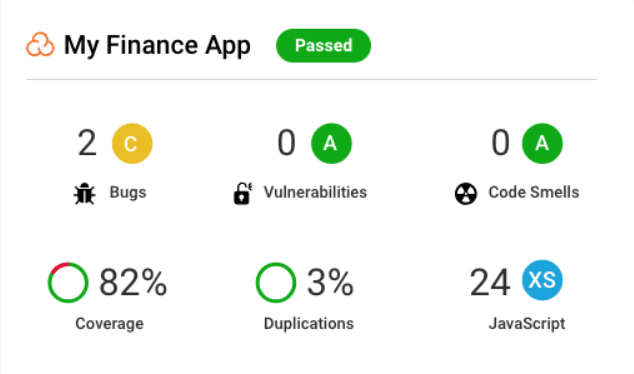
Permet de *détecter les vulnérabilités et le code de faible qualité*.



### Sonar Cloud :

Permet de *lier Sonar à vos projets GitHub* *et dans votre CI*.



## Méthodologies de travail :

### Traditional process model :

**Les étapes :**

* **Requirements :** on créé un *cahier des charges* en expliquant comment *répondre à une problématique bisness*.
* **Design :** on réfléchit à *comment implémenter* ce cas d’usage, on peut utiliser des *UML*.
* **Implementation :** on *implémente*.
* **Verification :** on *test* notre application *pour s’assurer que l’on répond bien aux besoins* et aux critères d’accéptances.
* **Maintenance :** on *maintient* notre *application*.
* **Deployment :** on *délivre* notre projet, c’est le *seul moment* *où l’on pourra obtenir des feedbacks de nos utilisateurs*.

**La problématique principale de cette approche :** c’est que *l’on ne délivre pas souvent* et donc *on n’aura pas de retours fréquents de nos clients*, on pourrait partir dans la mauvaise direction sans le savoir.

### Agile :

O*n délivre plus souvent et ont créé de plus petits projets livrables*, ce sont des itérations, ce qui permet *d’obtenir* beaucoup *plus de retours de nos utilisateurs* et le tout de façon plus fréquentes. Une itération s’appelle un sprint.

### Scrum :

**C’est quoi ?** C’est le modèle *Agile adapté* *pour* l’industrie *du développement*.

* Il y a une notion de *backlog* qui contient toutes les tâches à faire et des *sprints* dans lesquels on va *déterminer toutes les tâches prioritaires sur lequel on s’engage de délivrer* durant une itération.

**L’équipe :**

* **Product owner :** *responsable de prioriser* les tâches du livrable de chaque itération.
* **Scrum Master :** a.

**Exemple de stories :**

* **Fonctionnelles :** As a <common, registred, expert> user, I want …, so that I …
* **Techniques :** the system will run xxx platforms, the system will use azure to deliver scalability required, will respond in less than 2 seconds

#### Important

**Important :**

* Il est ultra *important de garantir une communication fréquente et régulière entre vos équipes et vos clients* afin de transformer leurs problèmes en solutions techniques.
* *Les réunions permettent de supprimer* les « impediments », *les points de blocages et d’incompréhension et rendent vos stories ready.* Les requirements doivent être revues par vos développeurs et s’assurer qu’ils comprennent ce qu’ils développement.
* Il faut *rajouter du temps alloué pour le refactoring* *et l’ajout de features techniques dans l’enveloppe prévu et durant le calcul de la vélocité*. On peut *vendre cette façon de travailler* en expliquant que *cela va réduire les coûts* *de maintenabilité en accélérant notre façon de travailler*.

**Trouver les requirements (les besoins) de nos utilisateurs :**

* On utilise notre *imagination*.
* On envoie des *questionnaires* ou on *interviews* *nos* *utilisateurs*.
* On *passe une journée dans leurs locaux en observation*, il n’y a rien de mieux pour mieux cerner leurs besoins.

**S’assurer que nos solutions sont adaptées à nos utilisateurs ?**

* On créé des *prototypes, des mocks ups, des maquettes pour leur donner un aperçu de ce que pourra donner la solution finale et déterminer si cela leur convient*.
* On *s’assure que la solution* que l’on propose *est* totalement *faisable*.
* On *communique et livre régulièrement pour récupérer leurs précieux retours.*

## Infrastructure

### Vocabulaire

**Scalability :** c’est la *capacité d’un système à continuer de bien fonctionner et de servir de bonnes performances en cas de plus forte demande*.

* **Vertical scaling / scale up :** changer le types de hardware, *CPU*, *RAM*, *storage*.
* **Horizontal scaling / scale out :** modifier le *nombre de serveurs*, *la charge de tous les serveurs doit automatiquement s’équilibrer*.

**Resilient :** c’est *la capacité d’un système à continuer de fonctionner en cas de panne*.

**Idempotent :** *processing the same operation several times has the same effect as processing it once*.

### Virtualisation / containeurization

**Containeurization :** technique qui permet que *votre application prenne avec elle toutes ses dépendances* *permettant qu’elles tournent depuis n’importe où*.

**Docker (runtime Container) :** *outil pour lancer des conteneurs*.

**Les registres :** *les images se déposent dans des registres*.

* **Public :** c’est u*ne sorte de npm registries pour les images les plus utilisées*.
* **Privée :** on les utilise *pour une entreprise*, *Azure propose un* composant agissant comme un registre privée.

**Volumes :** c’est le moyen de *persister des données générées et utilisées par vos conteneurs*.

**Pods :** c’est la *plus petite unité de déploiement* dans Kubernetes qui regroupe *un conteneur ou un ensemble de conteneurs qui doivent être déployés en même temps et qui partage un même réseau et système de fichiers*.

**Orchestrateurs (de conteneurs) :** logiciel qui permet de créer des clusteurs, c’est-à-dire choisir *le nombre de réplicas* de vos pods, les *load balancer* *et changer le CPU et la RAM de vos machines*.

* **Infrastructure as code :** permet de *gérer l’infrastructure* uniquement *par du code*, l’un des *avantages* c’est que la *configuration* est *modifiable à la volé* *et* que ces fichiers seront *versionnés*.

#### Docker :

##### A savoir

**A savoir :**

* *Les conteneurs s’instancient à partir d’images et se lancent depuis un runtime containe*r, tel que Docker *dans lesquelles elles seront lancées dans un* *environnement totalement isolé*.
* Les fichiers *DockerFile*, docker-*compose*.yml .*dockerignore* sont *versionnés*.
* Images -> containers -> pods -> orchestrator.

**Dockerfile VS Docker-compose.yml :** ces fichiers se situe à la racine de

* **Dockerfile :** ensemble de *commandes pour générer votre image* (FROM xxxilage, RUN dotnet build / publish / copy . . etc..).
* **Docker-compose.yml :** vos *ports* forwarding ? vos *variables d’environnements*, *les dépendances de votre application*, un *numéro de version*, vos volumes (les répertoires que vous utilisez pour persistez des données).
  + **Docker.compose.yml :** *configuration commune*.
  + **Docker.compose.<environment>.yml :** *configuration par environnement*.
* **.dockerignore :** ignorer des fichiers et des répertoires *durant l’appel des commandes ADD et COPY* de votre fichier Dockerfile.

##### Commandes et usage :

**Générer le Dockerfile et le Docker-compose.yml :**

* *Clic droit* sur votre projet > *Container orchestrator support* > *docker compose*.

**Publier une image dans votre registre privée :**

* Clic droit sur votre projet > publish > *azure docker registry*.

**Instancier et lancer un conteneur à partir d’une image :**

* docker-compose docker.compose.yml -f docker.compose.<environment>.yml -f

**Arrêter et supprimer vos conteneurs :**

* docker-compose docker.compose.yml -f docker.compose.<environment>.yml -f down

**Créer un registre privé pour stocker vos images Docker :** Azure > chercher « container registries » > add, *l’url de votre registre sera <registryname>.azurecr.io/<imagename>*.

**Récupérer une image provenant d’un registre en local (clone) :** docker pull <uri>:version1:0.

**Lister vos images sur votre machine locale / repository :** docker images [repository].

**Vous connectez à votre registre azure :** docker login <registryname>.azurecr.io.

**Assigner le chemin d’accès d’une image qu’elle aura sur azure :**

* docker image yourimage <registryname>.azurecr.io/randompath/imagename.

**Envoyez vos modifications sur votre registre :**

* docker push <registryname>.azurecr.io/randpath/imagename.

### Cloud :

#### Vue simplifiée

**Exemple de composants :** Azure Sql Server / Azure Wep App / Azure Cognitive Services / Azure Dev Ops / Kubernetes Orchestrator / Private Docker Registry / Azure Cosmos Db / Redis cache / Azure Fonction.

**IAAS / Infrastructure As A Service :**

* *Virtual machines* & machine learning.
* **Network :** *Load balancer*, virtual network, DNS Zones.
* **Storage :** *stockage de fichiers*.

**PAAS / Platform As A Service :**

* Azure *Web App*.
* Azure *Sql Server*.
* **Azure Cognitive Services :**
  + *Speech to text*.
  + *Translate text*.
  + Reconnaissance d’images.
  + **Exemple pour traduire une phrase, il faut que :**
    - *Créer un composant Azure Cognitive Services* sur Azure.
    - *Appeler ce service* via des appels *HTTP* pour traduire vos phrases.

**SAAS / Software As A Service :** Office 365, Azure Dev Ops.

**FAAS / Function As A Service / Serverless :** ça s’auto scale, c’est ultra bien pour un micro service ou lorsque l’on a qu’une petite fonctionnalité qui est appelé beaucoup plus que les autres.

#### Azure fonction :

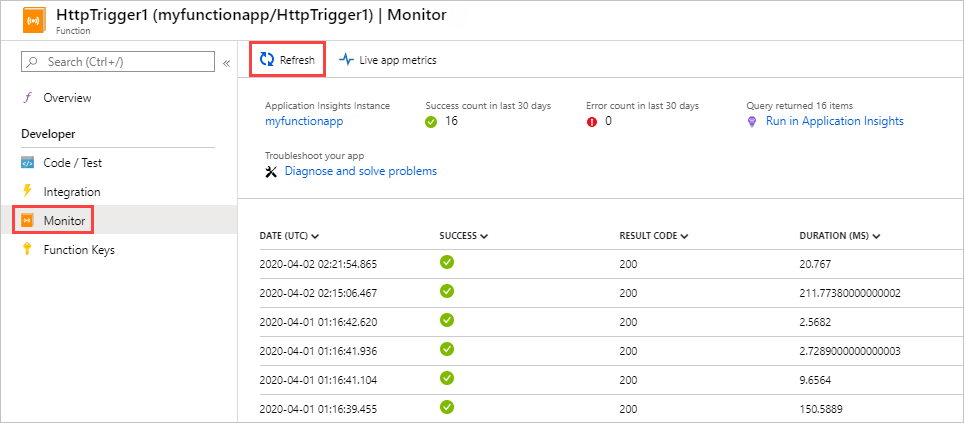
*Installer l’extension VS Azure Function and Web Jobs Tools et modifier l’installer de VS et cocher Azure Development.*

**A savoir :** scalable, resilient, se moniteur, templates.

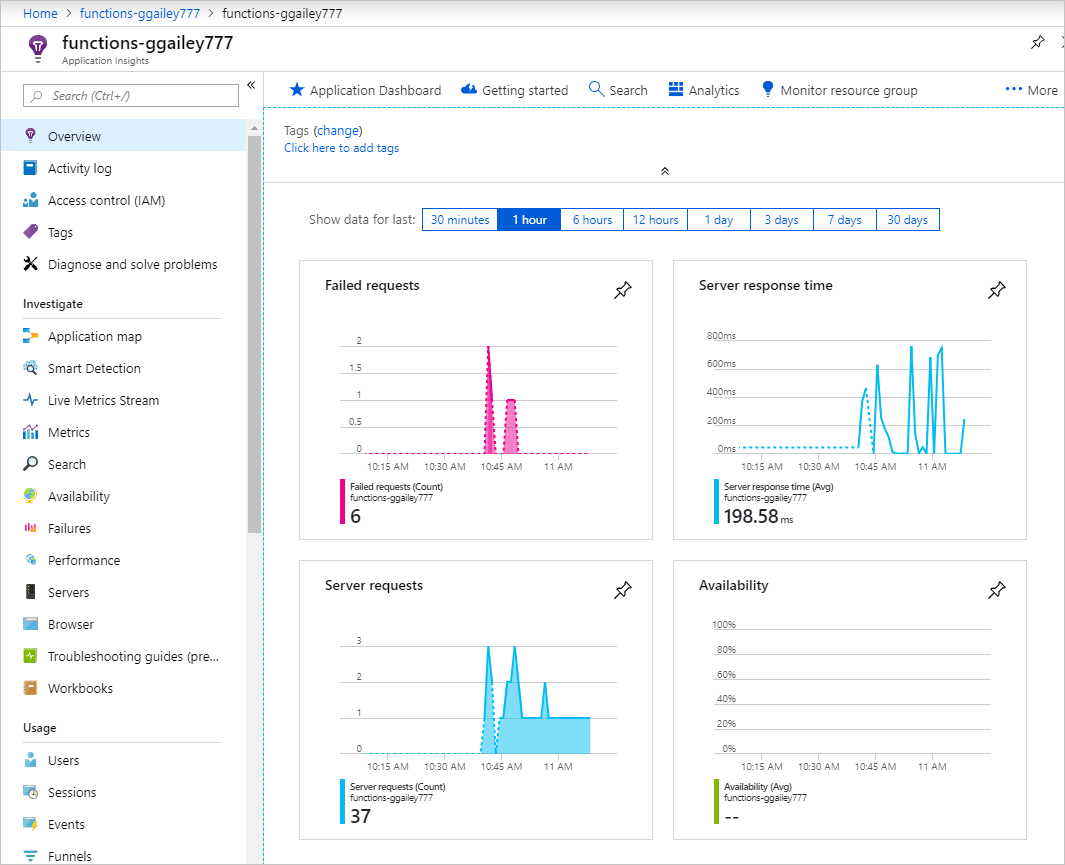
* Les fonctions azure proposent une *liste importante de modèles existants* *et peuvent-être écrites dans différents langages* de programmations.
* Vous pouvez *les éditer, les tester et les monitorer* depuis un éditeur sur Azure mais aussi les *débuguer et les créer depuis Visual Studio*.

**Les onglets :**

* **Moniteur :** date, has *success*, result code, *duration*.



* **Application Insights > overview :** nombre de requêtes, *nombre de requêtes échouées*, *évolution du temps de réponse,* taux de disponibilité.

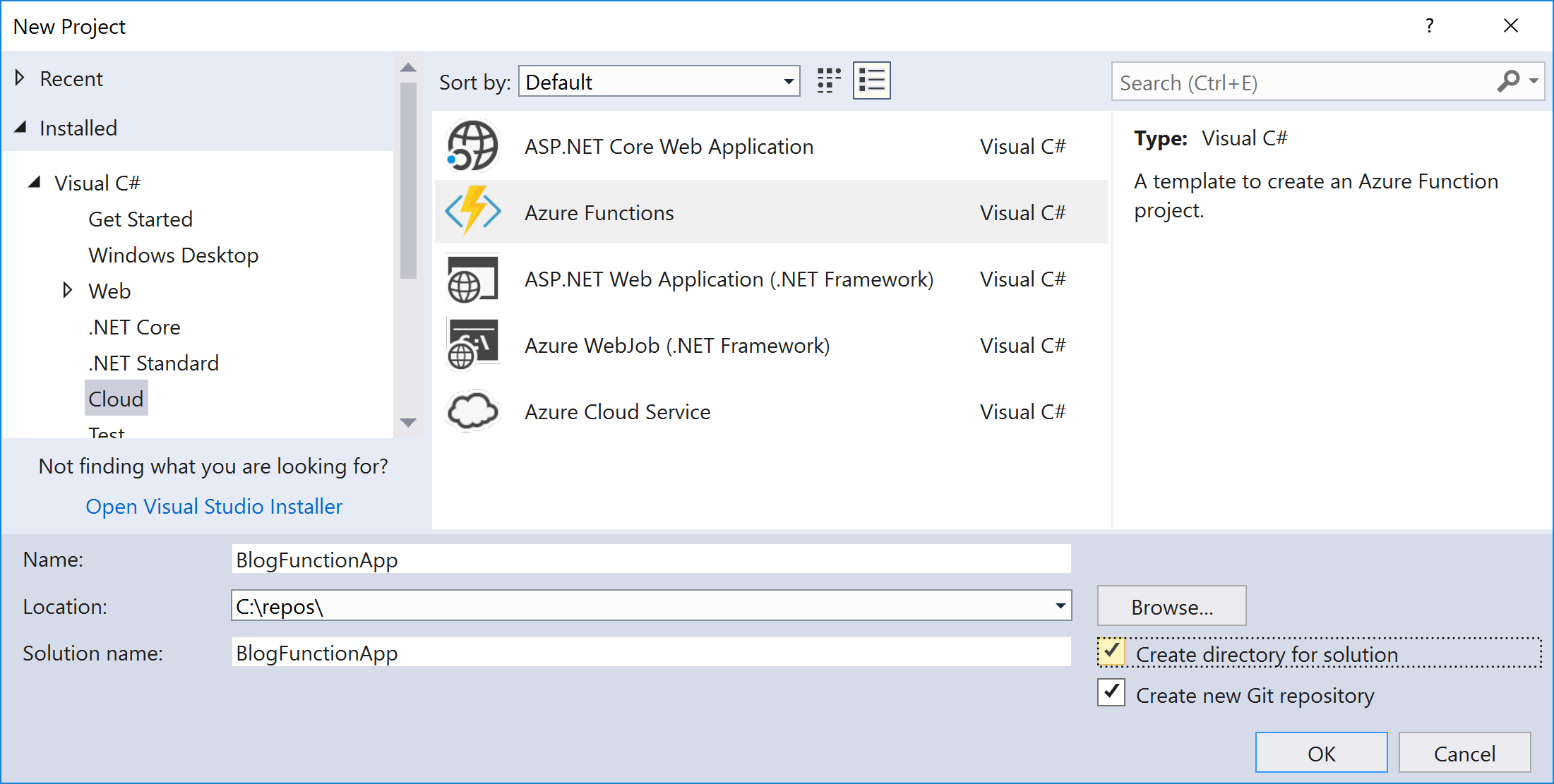


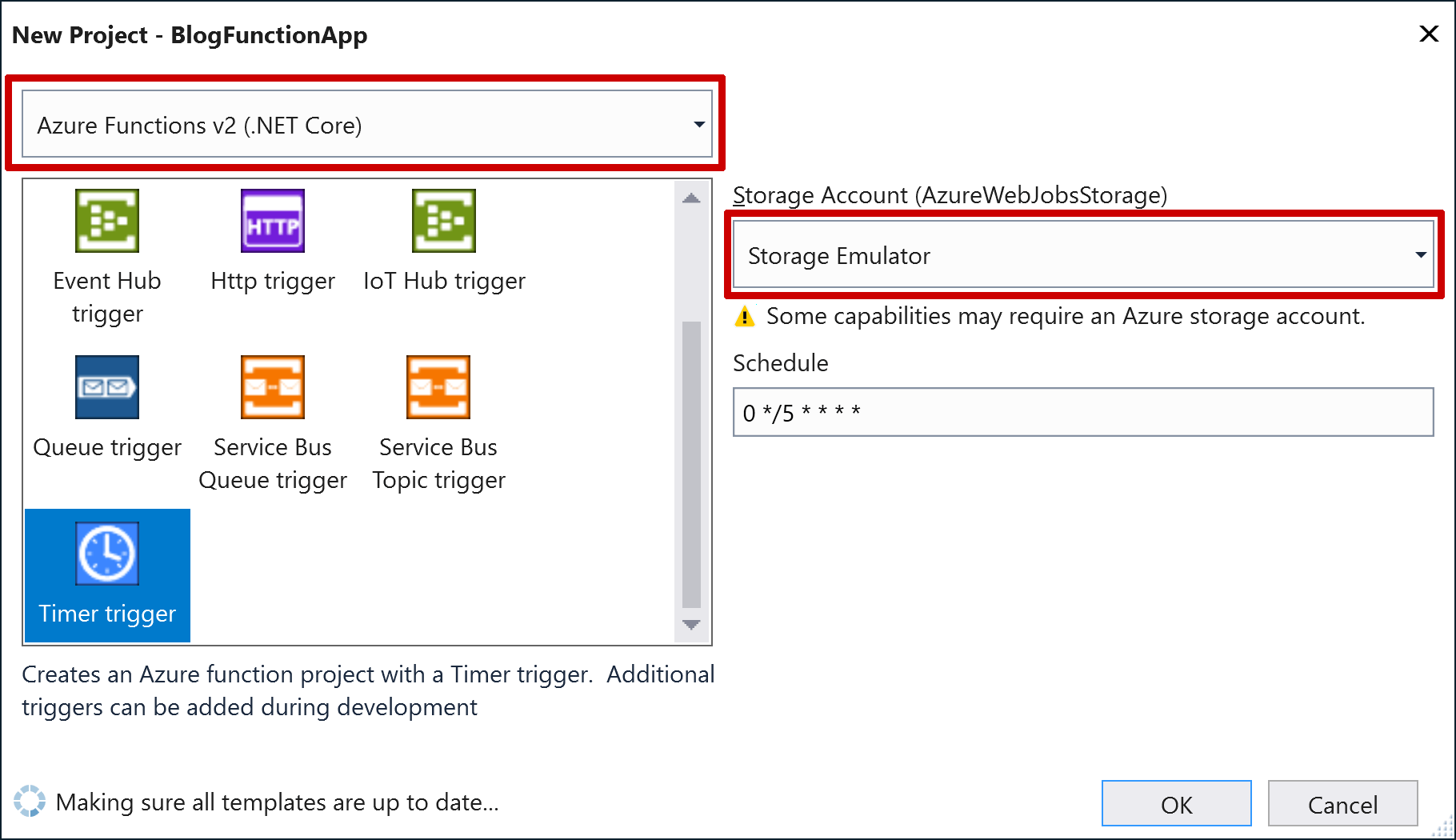
**Les templates :**

* **Timer :** fonction qui se lancer à intervalle régulier, c’est un *job schédulé* en quelque sorte.
* **Webhook :** un *évènement* web *que l’on peut déclencher et sur lequel on peut s’abonner*.
* **HTTP :** un API *endpoint*.

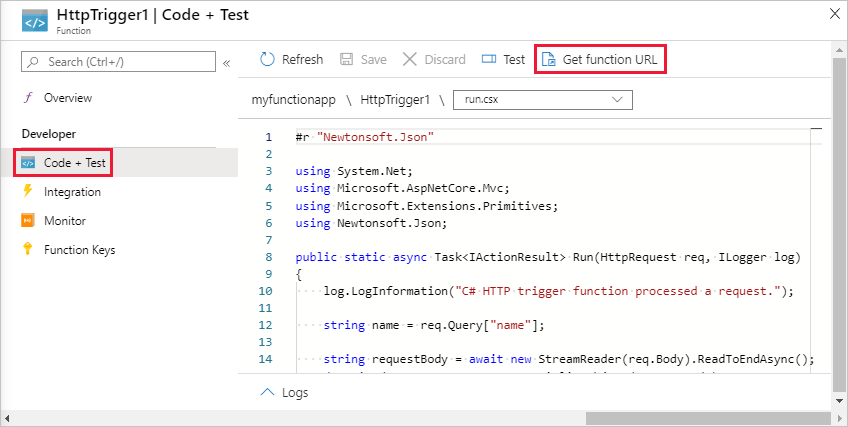
**Créer une Azure Fonction :**

* **Depuis Azure :** create a resource > function app > choisir un template.
* **Depuis VS (mieux pour tester et déboguer)** : *New project* > *cloud* > *azure functions*.





**Tester votre fonction Azure et récupérer son URL :**



## Architecture :

### DDD :

**A savoir :**

* **Résumé en quelques mots :** *bounded context*, language *ubiqueux*, *3 couches* (domaine, infrastructure, application), *events* en publisher/subscriber, *value objects*, *aggregate root*, *repository* & *unit of work*.

**DTO :** data transfer object, *c’est un objet qui est utilisé pour passer d’une couche à une autre* ?

#### Vocabulaire

#### Bounded Context :

**Bounded Context & Ubiquious Language :**

* L’application est découpée en plus petit domaines, *le contrat qu’implémente ces domaines s’appelle un Bounded Context*, c’est en quelque sorte une frontière logique.
* *Chaque Bounded Context parle son propre langage métier* qui est utilisé à la fois par les experts métier et les équipes de développement, c’est qu’on appelle le *langage ubiqueux*, *son utilisation facilite le partage de la connaissance métier et la communication* avec les experts du domaine.
* *On utilise des évènements pour communiquer avec d’autres bounded context*, ce qui réduit considérablement le couplage.
* Se retrouver avec des *noms d’agrégats similaires dans différents Bounded Context est normal* et vient du fait que chaque bounded context parle son propre langage métier.

#### Value objects :

**Les value objects d’identité :** AccountId, ContactId.

* Rend notre code *plus SOLID*.
* *Toutes leurs propriétés sont en readonly*.
* Ils *gèrent leur propre validation* en *appelant une méthode valide tous les champs* *à chaque* fois que votre objet est *muté* (Create, Update, Add).
* **Utiliser des values objects au sein de Entity Framework :**
  + Utiliser la méthode OwnsOne.
  + *Mapper vos champs readonly* (private setter).
  + NotMapped ?

#### Aggregate Root & Aggregate / DDD Entities :

**A savoir :**

* Une racine d’agrégat devrait *obéir aux règles des values objects*, c’est-à-dire n’avoir que des *champs en readonly*, *gérer sa propre validation*.

**Une racine d’agrégat et composé :**

* **Des propriétés suivantes :**
  + **Version :** elle est *incrémentée à chaque nouvelle mise à jour* au moment du *SaveChanges* par le design pattern *UnitOfWork*.
  + **Events :** à chaque mutation on *peut remonter un évènement*.
  + **Ma vision :** CreateBy, CreatedAt, CreatedOnBehalf, UpdatedBy, UpdatedAt, UpdatedOnBehalf.
* **Des méthodes suivantes :**
  + **Validate :** *vérifie l’intégralité* de vos propriétés.
  + **RaiseEvent :** *ajoute* un nouvel *évènement*.
  + *Une méthode par mutation* (Create, Update, Add, etc…) *qui appelle la Validate()* après avoir mis à jour ses propriétés.
  + **Ma vision :**
    - **Create() et Update() :** qui *mettent à jour les propriétés* CreateBy, CreatedAt, CreatedOnBehalf, UpdatedBy, UpdatedAt, UpdatedOnBehalf.
    - **Flush :** renvoit les évènements puis les vide, incrémente la version, et appelle les méthodes Create() et Update().

**Implémentation brouillon :**

public IEnumerable<Events> Flush(IAuthentificationContext authentificationContext)

{

\_version++;

if (\_createdBy is null)

Create (authentificationContext);

else

Update(authentificationContext);

var eventCopy = copy(events);

\_events.Clear();

return eventCopy;

}

private void Create(IAuthentificationContext authentificationContext)

{

\_createdBy = authentificationContext.UserId;

\_createdAt = DateTime.NowUtc;

\_createdOnBehalf = authentificationContext.ImpersonifiedUserId;

}

private void Update(IAuthentificationContext authentificationContext)

{

\_updatedBy = authentificationContext.UserId;

\_updatedAt = DateTime.NowUtc;

\_updatedOnBehalf = authentificationContext.ImpersonifiedUserId;

}

protected void RaiseEvent(IEvent event) => \_events.Add(event) ;

protected abstract void Validate();

#### Unit Of Work & Repository

**A savoir :**

* L’objet unit of work *permet de persister vos données sur votre base de données et représente l’identité de la transaction en cours*.
* Ce pattern va de pair avec le pattern Repository.

Une image contenant capture d’écran, oiseau

Description générée automatiquement

**Code brouillon :**

public Task<bool> SaveChanges()

{

var allEvents = \_dbContext.SelectAllAggregateRootModified().SelectMany(\_ => \_.Flush());

foreach (var event in allEvents)

\_mediator.SendEvent(event);

return \_dbContext.SaveChanges();

}

#### Repository : pas sûr du tout, à revoir…

**A savoir :**

* Cette classe semble *mutualiser les requêtes* en lecture et écriture *faite à votre base de données et vos contextes*.
* On peut avoir *un repository par racine d’agrégat* et *chaque repository se fait injecter UnitOfWork*.

Une image contenant couteau

Description générée automatiquement

**Notes à merger dans mes notes CQRS :** utilisé Sql Server pour l’écriture et CosmosDb ou ES pour la lecture.

### Microservices

**Quand doit-on faire du microservice ?**

* On a un *besoin de* *réduire le scope de notre application et le besoin d’y assigner une équipe dédiée.*
* Nous avons un « microservice / *bounded context* » qui est *appelé beaucoup plus que les autres* qui *nécessite un meilleur auto-scale.*
* Nous avons une *équipe composée* de plusieurs *séniors* *qui ont déjà une expérience* dans ce domaine.
* On souhaite *réduire considérablement le couplage* de nos applications.

**Les problématiques rajoutées par les microservices :**

* Nous allons avoir une *CI/CD plus complexe et plus cher*, on *va devoir livrer* beaucoup *plus* souvent.
* On va avoir *beaucoup plus de communication* réseaux ce qui *coûte cher*.
* On va devoir *rajouter* beaucoup *de nouveaux* *tests d’intégrations* pour s’assurer que tous fonctionnent.
* On va avoir *besoin de créer des clusteurs* pour augmenter notre taux de disponibilité, pffiiiou qu’est-ce que ça coûte cher en fait.
* On va devoir besoin d’encore mieux monitorer nos apps.

**Les couches de communications :**

* **API Gateway :** une couche de vos microservices qui s’occupe de *prendre soin du routing des requêtes entrantes et de les dispatcher vers les bons microservices*.
* **Event Bus :** une couche de *gestion des évènements envoyés et récupérés*, peut se faire via Azure Service Bus, RabbitMQ, NServiceBus avec une communication *asynchrone* en *publisher/susbsriber* sur un channel dédié.

**A savoir :**

* *Un microservice n’a pas la permission d’accéder à la base de données ou le système de fichiers d’un autre microservice*, il est censé être owner.

## Fournisseur de données

### Les types de fournisseurs de données :

**Les types de base de données :**

* **Base de données relationnel** : Sql Server, on *gère des collections d’objets* liés avec vos autres tables par des *clés étrangères*, cela coûte une blinde, bien pour les *opérations* *d’écriture*.
* **SGDB NoSql :** Azure Cosmos Db, le *schéma* de vos records ne sont *pas structuré*, *vous stockez* en base *des documents* qui sont en fait *des graphes d’objets complexes un peu à la sauce ES*, il n’y a *pas de clés étrangères*, les requêtes sont écrites en JSON, c’est censé *coûté pas cher* et offre souvent *de meilleures performances en lecture surtout si vous dénormalisez vos données.*
* **Moteur de recherche textuelle :** Elastic search, ça *ressemble à du NoSql* (schéma non structuré, documents, graphe d’objets, pas de clés étrangères, requêtes en JSON) *mais l’indexation semble très différente*, bien pour de la *lecture*.

### Entity Framework :

### A savoir :

* La *requête est exécutée* sur votre IQueryable *au moment où vous récupérez les données*, c’est-à-dire au moment d’un ToList, ToArray ou d’un First.
* La méthode *SaveChanges applique vos changements au sein d’une transaction*.

**Exécuter une requête SQL :**

* **Renvoit rien :** int DbContext.Database.ExecuteSqlRaw(string sql, prams object).
* **Renvoit vos records :** context.yourDbset.FromSqlRaw(string sql, params parameters) ;
  + Ex : var clients = context.Clients.FromSqlRaw(« SELECT \* FROM Clients WHERE x = (0)”, x)
  + Il est conseillé de *stocker vos requêtes dans des fichiers* / resources.

### Configuration :

* **Le CLI :**
  + **Pour la console de Visual Studio :** Microsoft.EntityFramework.Tools.
  + **Pour la console de Powerhell :** Microsoft.EntityFramework.Design.
* **Appliquer toutes les migrations dans le code :** context.Database.Migrate().

#### Les migrations :

Chacune de vos migrations contiennent une méthode Up *qui exécute la migration* et une méthode Down qui *permet d’annuler les effets de votre migration*.

**Créer une migration :**

* **Console de VS :** Add-Migration <name>.
* **Powershell :** dotnet ef migrations add <name>.

**Appliquer une migration :**

* **Console de VS :** Update-Database.
* **Powershell :** dotnet ef database update.
* **Appliquer toutes les migrations dans le code :** context.Database.Migrate().

**Executer une requête SQL dans une migration :**

* Créer une *migration vide* en faisant un Add-Migration.
* Appeler migrationBuilder.Sql *depuis* vos méthodes *Up et Down*.

## API :

**A savoir :**

* *Utiliser des objets pour éviter les breaking changes*, exemple.

class xDto { string address }

class xDto { Address address }

**Status code :**

* **202 accepted :** la création a été *acceptée mais* sera *différée*.
* 301 moved permanently.
* 307 moved temporarily.
* **401 unautorized :** *requière une authentification*.
* **403 forbidden :** *authentifié mais requière des droits plus élevés*.

**Verbes HTTP :**

* PUT = *indempotent*.
* PATCH = pas indempotent car il peut y avoir des add / remove.

### Swagger

**OpenApi :** Un *standard pour décrire les API REST*.

**Swagger :** framework qui *implémente OpenApi*.

#### Configuration

**Activer la documentation de vos endpoints par triple slash :**

/// <response code = »201>XXx have been created</response>

* **Mettre à jour le csproj :**

<PropertyGroup>

<GenerateDocumentationFile>true</GenerateDocumentationFile>

<NoWarn>$(NoWarn);1591</NoWarn>

</PropertyGroup>

## .NET CORE

### A savoir

**A savoir :**

* **Properties/launchSettings.json :** *comment lancer votre application depuis Visual Studio* (tests).
* **Partager des informations dans vos middleware :** HttpContext.Items.

**Attributs sur les paramètres de vos endpoints :**

* **[FromQuery] :** les paramètres proviennent d’une query string (*les* *paramètres de l’url*).
* **[FromHeader] :** les paramètres proviennent d’un *entête*.
* **[FromServices] :** *injecte une dépendance en paramètre*, il vaut mieux que votre dépendance ne soit pas utilisée ailleurs dans votre classe, sinon il vaudra probablement mieux l’injecter depuis le constructeur.
* **[FromForm] :** les paramètres proviennent d’un *formulaire*.
* **[FromRoute] :** valeur par *défaut*.

**Le routing :** UseEndpoints(endpoints =>

endpoints.MapControllerRoute(“default”, “{controller=Home}/{action=Index}/?id”);

endpoints.MapGet(“hello/{country}”, context => context.Response.WriteAsync($”country is {context.GetRouteValue(“country”)}”))

.RequiredAuthorization(new AuthorizationAttribute() { Roles = “Admin” });

Endpoints.MapHub<ChatHub>(“/chat”);

);

**Activer l’HTTPS sur votre site :**

If (env.IsProduction()) app.UseHsts();

App.UseHttpsRedirection();

### Authentification :

**Authentification par :**

* **Clé d’API (secret) :** c’est une *permission d’utiliser votre API*, seulement le back et le front doivent avoir connaissance de cette clé, il faudra *utiliser du HTTPS* pour éviter que votre clé se fasse voler.
  + **Header :** Authorization : Api-Key <your key>.
* **Bearer Token :** vous avez *besoin d’une authentification* *ou de droits* (authorization) *pour* être permis d’*utiliser vos endpoints*. Le *jeton contient les claims* de l’utilisateur et donc les nécessaires pour calculer les droits de vos utilisateurs (role / policy, etc…).
  + **Header** :Authorization : Bearer <your token>.

**Implémentation brouillon basé sur :**

* **Les rôles :**

If (User.IsInRole(“xrole”))

[Authorize(Roles = « xrole,otherole”)]

* **Une politique et le contenu des claims :**

AddAuthorization(options => options.AddPolicy(“xpolicy”, p => p.RequireAssertion(c => c.User.HasClaim(c => c.xClaim = “t”)));

### API :

**A savoir :**

* **Retours de vos endpoints :** OK, Created, Accepted, BadRequest, Unauthorized, Forbid.
* *Vos contrôleurs d’API doivent hériter de ControlerBase*.

### MVC

#### A savoir

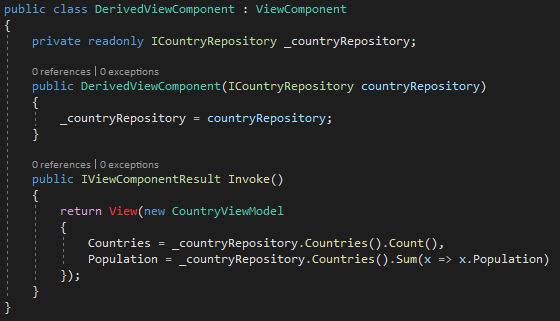
**Les retours des actions (IActionResult) :**

|  |  |
| --- | --- |
| Type de retour et description | Code |
| ViewResult : renvoyer une *vue*. | return View(viewName, viewModel) ; |
| RedirectResult : *rediriger* *vers* une autre *action* / endpoint. | return LocalRedirect(xx)  return RedirectToAction(actionName, controllerName, new {xParam1 }) ; |
| ContentResult : du *texte en spécifiant son MIME type*. | return Content(yourText, yourMimeType) ; |
| FileResult : un fichier. | return File(stream, mimeType) ; |

#### Les views :

**Les types de vues :**

* **View :** un *template HTTP* (*cshtml*), elles *dépendent* d’une classe *de données qui s’appelle un view model*.
* **Partial view :** ce sont des *vues partagées et réutilisées par vos autres vues*.
* **View component / les pages :** c’est une *vue partielle*, *elles dépendent d’un objet anonyme et non d’un view model et view data* ce qui rend leur code plus prédictible. Elles ont un *fichier dédié ce qui mieux organisé*.



* **Layout view :** c’est une vue qui agit comme un calque qui *contient les éléments de vos pages qui sont très récurrent* (balise html, body, head, header & footer, nav bar).

Une image contenant capture d’écran

Description générée automatiquement

**Où se trouvent vos vues :**

* **Les views :** Views/{controllername}.
* **Les views components :** Pages/Components/{componentname}/.

Une image contenant capture d’écran

Description générée automatiquement

* **Les vues partielles et vos calques :** Views/Shared/.

Une image contenant capture d’écran

Description générée automatiquement

#### Globalization de vos vues :

**A savoir :**

* On peut *changer de langue* *avec l’entête Accept-Language*.
* Le *cookie* nommé *AspNetCore.Culture contient la localization de l’utilisateur*.
* *Vos* *fichiers de localization* sont des ressources (.resx) contiennent un *dictionnaire* de clé valeurs.

**Configuration :**

App.UseRequestLocalization(new ReqiestLocalizationOptions

{

DefaultRequestCulture = new RequestCulture(“en”, “en »), // localization du format des dates et des nombres, puis de vos strings.

SupportedCultures = new CultureInfo[]{} // *localization du format de dates et des nombres*.

SupportedUICultures = … // localization des strings

FallBackToParentCultures = true // si l’utilisateur change de langue mais qu’elle n’est pas contenue par SupportedCultures et SupportedUICultures alors celle par DefaultRequestCulture sera prise.

FallBackToParentUICultures

});

Services.AddLocalization(options => options.ResourcePath = “Ressources”);

// Exemple : Views/{controllername}/Index<culture name>.resx.

**Localizer une chaîne depuis vos vues :**

@inject ViewLocalizer Localizer

Localizer[« key”]

#### Blazor :

@model MyViewModel // *spécifier notre modèle de données* (view model).

@using Newtonsoft // *inclure un fichier*.

@inject IViewLocalizer Localizer // *injecter* une *dépendance*.

@{

Layout = « \_MyLayout » // *définir le calque* qui nous englobe.

}

@\* *comment* \*@

<div>@Model.PropertyName</div> // *afficher une variable*.

<div>@DateTime.NowUtc.ToString()</div> // afficher le retour d’un code.

@{ // code csharp.

var x = 3 ;

}

@if (true) {} // *flow control*.

else

@while (true)

@for (var I = 0; I < 3; i++)

@foreach (var x in mylist)

<form asp-controller=”controllerName” asp-action=”actionName”> // *formulaires*.

<div asp-validation-summary></div> // *afficher toutes les erreurs de validation* ici.

<input type= »text » asp-for”Address.Town”> // *Lier une propriété du view model au contenu du champ*

<div data-valmsg-for= »Address.Town »></div> // *validation par champ*.

// *alternative pour envoyer des données à vos vues* sans passer par le viewModel.

@{ ViewData[« key »] = value ; } // définir une valeur.

@ViewData[« key »] // afficher la valeur.

// *Afficher une vue* / *view component.*

<partial name= »partialViewName » for=”Model.SubModel” ou model=new Model{}/>

@await Component.InvokeAsync(viewComponentName, new {});

<environment include= »Staging »> </environment> // *code différent par environnement*.

Url.Action(actionName; controllerName, new { … }) // *calculer une url*.

#### Tester ASP NET MVC :

**Les librairies :**

* **Microsoft.AspNetCore.Mvc.Testing :** tester le site.
* **AngleSharp :** parser les pages et le DOM.

**Tester les erreurs de validation :**

Var controller = new TotoController();

Controller.AddModelError(propertyName, errorMessage);

IViewResult result = controller.YourAction(yourMock.Object);

Assert.Equal(result, …);

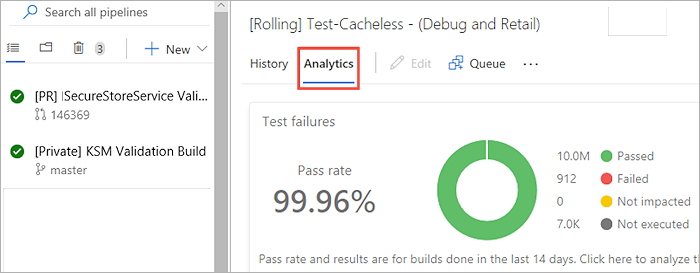
## Dev Ops

**A savoir :**

* *Dev Ops* est représenté par le symbol de l’infini qui *représente* le cycle de développement qui tourne en *boucle dans lequel* *on plannifie, build, integre, déploie, récupère les retours utilisateurs*.
* La CI *permet d’identifier les problèmes plus rapidement mais aussi de protéger et de stabiliser votre branche master* *à chaque nouveau commit*.

### Azure Dev Ops :

**Affiche des metrics sur toutes vos builds :** *History* > *analytics*.



**Afficher le résultat d’une build :** *History* > *sélectionner* votre build > *tests*.

Une image contenant capture d’écran

Description générée automatiquement